Chapter 8. Sorting in Linear Time

8.1-1.
The best case occurs when the array is already sorted, where there are $n-1$ pairs of relative ordering, therefore the smallest possible depth of a leaf is $n-1$.

8.1-2.

$$\lg(n!) = \sum_{k=1}^{n} \lg k < \sum_{k=1}^{n} \lg n = n \lg n \Rightarrow \lg(n!) = O(n \lg n).$$

$$\lg(n!) = \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} \lg k + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} \lg k > \sum_{k=1}^{\lfloor \frac{n}{2} \rfloor} 1 + \sum_{k=\lfloor \frac{n}{2} \rfloor + 1}^{n} \lg \frac{n}{2} = \frac{n}{2} + \frac{n}{2}(\lg n - 1) \Rightarrow \lg(n!) = \Omega(n \lg n).$$

$$\Rightarrow \lg(n!) = \Theta(n \lg n).$$

8.1-3.
Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the $n!$ permutations of the input appears as some leaf, we have $n! \le l$. Since a binary tree of height h has no more than 2^h leaves, we have $n! \le l \le 2^h$. Thus $h \ge \lg(n!) = \Theta(n \lg n)$.
Since $\lg(\frac{n!}{2}) = \Theta(n \lg n) - 1 = \Theta(n \lg n)$, there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n.
Since $\lg(\frac{n!}{n}) = \Theta(n \lg n) - \lg n = \Theta(n \lg n)$, there is no comparison sort whose running time is linear for a fraction of $\frac{1}{n}$ of the $n!$ inputs of length n.
Since $\lg(\frac{n!}{2^n}) = \Theta(n \lg n) - n = \Theta(n \lg n)$, there is no comparison sort whose running time is linear for a fraction of $\frac{1}{2^n}$ of the $n!$ inputs of length n.

8.1-4.
Since each subsequence has length of k, there are $(k!)^{\frac{n}{k}}$ possible permutations. The decision tree with height h has the property of $2^h \ge (k!)^{\frac{n}{k}}$, which

leads to $h \geq \frac{n}{k}\lg(k!) = \Theta(k \lg k)\frac{n}{k} = \Theta(n \lg k) \Rightarrow h = \Omega(n \lg k)$.

8.2-1.
$[2] \rightarrow [2,3] \rightarrow [1,2,3] \rightarrow [1,2,3,6] \rightarrow [1,2,3,4,6] \rightarrow [1,2,3,3,4,6] \rightarrow$
$[1,1,2,3,3,4,6] \rightarrow [0,1,1,2,3,3,4,6] \rightarrow [0,1,1,2,2,3,3,4,6] \rightarrow [0,0,1,1,2,2,3,3,4,6] \rightarrow$
$[0,0,1,1,2,2,3,3,4,6,6]$

8.2-2.
Suppose two position $i < j$ have an identical element k. Since $j > i$, the loop iterates over $A[j]$ before $A[i]$, and in that iteration $C[k]$ is decremented by 1. Since C is never incremented, we can ensure that $C[k]$ is decreased when the loop iterates over $A[i]$, so it preserves the order of tied elements.

8.2-3.
The algorithm still works correctly, because the modification only affects the order of elements with the same key.
The modified algorithm is not stable, because the order of elements with the same key is flipped.

8.2-4.
The preprocessing is exactly same with those of counting sort, so that C[i] is the number of elements less than or equal to i of the n integers. Computing C[b] - C[a-1] gives the number of integers fall into a range $[a, b]$, which has $O(1)$ time.

8.3-1.
$SEA, TEA, MOB, TAB, DOG, RUG, DIG, BIG, BAR, EAR, TAR, COW, ROW, NOW, BOX$
$TAB, BAR, EAR, TAR, SEA, TEA, DIG, BIG, MOB, DOG, COW, ROW, NOW, BOX, FOX$
$BAR, BIG, BOX, COW, DIG, DOG, EAR, FOX, MOB, NOW, ROW, RUG, SEA, TAB, TA$

8.3-2.
Insertion sort and merge sort are stable, whereas heapsort and quicksort are not. We can make any sorting algorithm stable by adding an additional index of the elements to each element, and use this as a tiebreaker.

8.3-3.
The loop invariant is: At the beginning of the each iteration of the loop, the array is sorted on the last $i - 1$ digits.

Initialization: Trivial. $(i-1=0)$
Maintenance: Since the subroutine is a stable sort and we're sorting on the ith digit, we get the result of the array which is sorted on the last i digits.
Termination: The loop terminates when the array is completely sorted.

8.3-4.
Converting them into base n and using the radix sort gives $O(n)$ computation time.

8.3-5.
d passes are enough to sort d-digit decimal cards.
Since we are dealing numbers with base 10, the number of piles of cards needed to be kept track is 10.

8.4-1.
$[0.13, 0.16, 0.20, 0.39, 0.42, 0.53, 0.64, 0.79, 0.71, 0.89]$
$[0.13, 0.16, 0.20, 0.39, 0.42, 0.53, 0.64, 0.71, 0.79, 0.89]$

8.4-2.
The worst case occurs when all keys fall in the same bucket and originally stored in the reverse order, which requires the traditional insertion sort which has $\Theta(n^2)$.
We can change the subroutine to heapsort or merge sort, which has $O(n \lg n)$.

8.4-3.
$\mathbb{E}[X^2] = 2^2 \cdot \frac{1}{4} + 1^2 \cdot \frac{1}{2} + 0^2 \cdot \frac{1}{4} = 1.5$
$\mathbb{E}^2[X] = \mathbb{E}[X]^2 = (2 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4})^2 = 1$

8.4-4.
Design buckets to be a circle centered at the origin with radius $\sqrt{\frac{i}{n}}$ for $i = 1, \cdots, n$, respectively. Then the probability that the point falls into the ith bucket is $\frac{\pi\frac{i}{n} - \pi\frac{i-1}{n}}{\pi \cdot 1^2} = \frac{1}{n}$, which is equivalent to bucket sort.

8.4-5.
We split the whole support of the probability distribution into n subintervals with each having probability mass $\frac{1}{n}$. Running bucket sort takes linear time.

8-1.

a. There are $n!$ possible permutations because all elements are distinct. Since all permutations are equally likely, each occurs with probability $\frac{1}{n!}$, corresponds to different leaf.

b. $D(T) = \sum_{i \in LT} D_T(l) + \sum_{i \in RT} D_T(l) = \sum_{i \in LT}(D_{LT}(l)+1) + \sum_{i \in RT}(D_{RT}(l)+1) = D(LT) + D(RT) + k$.

c. Let T be a decision tree with k leaves that achieves the minimum. Let i_0 be the number of leaves in LT, then $k - i_0$ is the number of leaves in RT. Then $d(k) = D(T) = D(LT) + D(RT) + k = d(i_0) + d(k - i_0) + k$, because LT and RT are decision trees with i_0 and $k - i_0$ leaves that minimizes the value of D respectively. If not, then there exists some other LT or RT with i_0 and $k - i_0$ leaves which has smaller value of $D(LT)$ or $D(RT)$, but then $D(T)$ can be smaller by replacing LT or RT with the new argmin, a contradiction. Since T is chosen such that it minimizes D(T) among decision trees with k leaves, we have $d(k) = \min_{1 \le i \le k-1}(d(i) + d(k-i) + k)$.

d. Let $f(i) = i \lg i + (k-i)\lg(k-i)$ then $f'(i) = \frac{\ln i - \ln(k-i)}{\ln 2} = 0 \Leftrightarrow i = \frac{k}{2}$. We prove that $d(k) = \Omega(k \lg k)$ by induction. If $d(k) = \Omega(k \lg k)$ holds for $k < n$, then $d(n) = \min_{1 \le k \le n-1}(d(k) + d(n-k) + n) = \min(\Omega(k \lg k) + \Omega((n-k)\lg k) + n) = \Omega(2\frac{n}{2}\lg n2) + n = \Omega(n \lg n)$.

e. Since T_A needs at least $n!$ leaves, $D(T_A) \ge d(T_A) = \Omega(n! \lg(n!))$. The average run time is the weighted average depth of a leaf, and since each leaf occurs equally likely, the average run time is $\frac{n! \lg(n!)}{n!} = \lg(n!) = \Omega(n \lg n)$.

f. For any randomized comparison sort B, there is a deterministic comparison sort that has made B's random choices in advance. The result is a decision tree with number of choices less than or equal to the one of B.

8-2.

a. Counting sort.

b. Quicksort.

c. Insertion sort.

d. (a) : Yes, because it is stable and runs in linear time.

(b) : No, because it is not stable.

(c) : No, because it is runs in quadratic time, making the whole running time quadratic.

e.

Algorithm 1: IN-PLACE-COUNTING-SORT(A, k)

```
let $C[0, \cdots, k]$ be a new array
for $i = 1$ to $k$ do
    $C[i] = 0;$
end
for $j = 1$ to $A.length$ do
    $C[A[j]] += 1;$
end
for $i = 2$ to $k$ do
    $C[i] += C[i-1];$
end
$B = C[0, \cdots, k];$
$i = 1;$
while $i \leq A.length$ do
    if $B[A[i] - 1] \leq i$ and $i < B[A[i]]$ then
        $i += 1;$
    else
        swap $A[i]$, $A[C[A[i]] - 1];$
        $C[A[i]] -= 1;$
    end
end
```

This is in-place but not stable.

8-3.
a. Split integers by their number of digits and do bucket sort with the number of digits. Sort each bucket using radix sort. Join sorted subsets.
b. Split words by their first character and do bucket sort with their first character. Sort each bucket recursively using bucket sort.

8-4.
a. For all red jug, find blue jug that hold the same amount of water and group them. This is $\Theta(n^2)$.
b. The number of comparisons that the algorithm makes is the height of the leaf node from the decision tree. Since each comparison have 3 outcomes, each node has 3 children and the whole tree have $n!$ leaf nodes. The height is at least $\log_3(n!) = \Omega(n \lg n)$.
c. The algorithm similar to randomized quicksort do the job. The bound is

$O(n \lg n)$ and the worst case number of comparison is $\Theta(n^2)$.

8-5.
a. The ordinary meaning of sorting is equal to 1-sorted.
b. 2, 1, 3, 4, 5, 6, 7, 8, 9, 10
c.
$$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k} \Leftrightarrow A[i] \leq A[i+k].$$
d. Split the array into k parts so that the elements of i-th part has the index of $mk+i$. Then sort each part using quicksort, to sort in $O(\frac{n}{k} \lg \frac{n}{k})$ time. Therefore the total running time is $kO(\frac{n}{k} \lg \frac{n}{k}) = O(n \lg \frac{n}{k})$.
e. Sorting k-sorted array is equal to merging k sorted arrays into one sorted array. This is equivalent to 6.5-9, which has $O(n \lg k)$ time algorithm. f. The lower bound of sorting is $\Omega(n \lg \frac{n}{k})$, but since k is constant, this becomes $\Omega(n \lg n)$.

8-6.
a. Dividing 2n numbers to two lists with n numbers and sorting them solves the problem. There are $\binom{2n}{n}$ possible combinations.
b. Let h be the decision tree of the comparison sort that merges two sorted lists. Since there are at least $\binom{2n}{n}$ leaves, we have $2^h \geq \binom{2n}{n}$. Therefore, $h \geq \lg \frac{(2n)!}{n!n!} = \lg(2n!) - 2\lg(n!) = \Theta(2n \lg 2n) - 2\Theta(n \lg n) = \Theta(2n \lg(2n - n)) = \Theta(2n)$.
c, If they are not compared, then since there is no element between them, we cannot conclude the order of two elements.
d. Split the whole list two two sorted list $A = 1, \cdots, 2n-1$, $B = 2, \cdots, 2n$. We must compare 1-2, 2-3, $\cdots$, and $(2n-1)-(2n)$, total of $2n-1$ comparisons.

8-7.
a. Since $A[p]$ is the smallest misplaced element, and $A[q]$ is mlsplaced, so $A[q] > A[p]$, therefore $B[p] = 0, B[q] = 1$.
b. If N is an oblivious compare-exchange algorithm, then we have that for an arbitrary monotonous mapping, the order it applies on the compare-exchhange algorithm does not change the result. Since B is a monotonous mapping and $q < p$, otherwise $A[q]$ would be the smallest misplaced element, we have $B[q] > B[p]$ and $N(B[q]) > N(B[p])$, which means that N cannot sort B.

c. The odd-numbered sorting steps can be implemented by using an oblivious compare-exchange algorithm. The even-numbered steps only care about the sorted results of the odd-numbered sorting steps, not the details of how they are sorted, so columnsort can be treated as an oblivious compare-exchange algorithm.
d. After step 1, each column is a sequence of 0s followed by a sequence of 1s. In step 2, since $s|r$, each column will map to $\frac{r}{s}$ rows, and only one of those rows will be dirty and the rest will be clean. In step 3, clean rows of 0s wil move upwards and clean rows of 1s will move downwards. The number of dirty rows does not increase during the step 3. Therefore, afterwards, there will be at most s dirty rows between them.
e. After step 4, since the number of elements in the s dirty rows is s^2 and they are gathered, the dirty rows will map to a sequence (which read in column-major order) of length s^2. All other areas would be clean.
f. The dirty area is at most half a column in size. If the dirty area fits in a single column, then it will be cleaned in the step 5 and steps 6-8 will not corrupt it. If the dirty area lies over two columns, then it would be in the bottom half of the one column and the top half of the next. In step 5, this property is unchanged, and step 6 gathers the dirty area into a single column. Step 7 sorts the dirty area into the clean one, and step 8 rearranges all elements in the correct order.
g. If s does not divide r, then after step 1, the rows can contain $1 \to 0$ transitions as well, so the number of dirty rows after step 3 is at most $2s-1$. $r \geq 2(2s-1)^2$ ensures the correctedness of columnsort.
h. The columnsort do not need the assumption that s divides r, without any change in step 1. After step 2, let X be the set of dirty rows with only one $0 \to 1$ transition, Y be the set of dirty rows with only one $1 \to 0$ transition, and Z be the set of all other dirty rows. We claim that $\max(|X|, |Y|) + |Z| \leq s$. Since every row in X and Z contains $0 \to 1$ transition, we have $|X| + |Z| \leq s$. Similarly, every row in Y and Z contains $1 \to 0$ transition, we have $|Y| + |Z| \leq s-1$. Combining these two inequalities proves the claim. Now, after step 3, the clean rows of 0s move upwards and the clean rows of 1s move downwards. Consider $\min(|X|, |Y|)$ pairs of rows in which one of them is in X and the other row is in Y. Step 3 cleans at least one row among them, so at least $\min(|X|, |Y|)$ dirty rows are cleaned. The number of dirty rows remanining after step 3 would be at most $|X| + |Y| - \min(|X|, |Y|) + |Z| \leq s$. Therefore, the assumption for step 4-8 is same even without the divisibility restriction.