8 Sorting in Linear Time

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?

For a permutation $a_1 \le a_2 \le \ldots \le a_n$, there are $n-1$ pairs of relative ordering, thus the smallest possible depth is $n-1$.

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.

$$\begin{aligned}\sum_{k=1}^{n} \lg k &\le \sum_{k=1}^{n} \lg n \\ &= n \lg n.\end{aligned}$$

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

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 $1/2^n$?

Consider a decision tree of height h with r reachable leaves corresponding to a comparison sort on n elements. From **Theorem 8.1**, We have $n!/2 \le n! \le r \le 2^h$. By taking logarithms, we have

$$h \ge \lg(n!/2) = \lg(n!) - 1 = \Theta(n \lg n) - 1 = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for at least half of the $n!$ inputs of length n.

Consider the $1/n$ of inputs of length n condition. we have $(1/n)n! \le n! \le r \le 2^h$. By taking logarithms, we have

$$h \geq \lg(n!/n) = \lg(n!) - \lg n = \Theta(n \lg n) - \lg n = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for $1/n$ of the $n!$ inputs of length n.

Consider the $1/2^n$ of inputs of length n condition. we have $(1/2^n)n! \leq n! \leq r \leq 2^h$. By taking logarithms, we have

$$h \geq \lg(n!/2^n) = \lg(n!) - n = \Theta(n \lg n) - n = \Theta(n \lg n).$$

From the equation above, there is no comparison sort whose running time is linear for $1/2^n$ of the $n!$ inputs of length 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.)

Assume that we need to construct a binary decision tree to represent comparisons. Since length of each subsequece is k, there are $(k!)^{n/k}$ possible output permutations. To compute the height h of the decision tree, we must have $(k!)^{n/k} \leq 2^h$. Taking logs on both sides, we know that

$$h \geq \frac{n}{k} \times \lg(k!) \geq \frac{n}{k} \times \left(\frac{k \ln k - k}{\ln 2} \right) = \frac{n \ln k - n}{\ln 2} = \Omega(n \lg k).$$

8.2 Counting sort

8.2-1

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

We have that $C = \langle 2, 4, 6, 8, 9, 9, 11 \rangle$. Then, after successive iterations of the loop on lines 10-12, we have

$$\begin{aligned} B &= \langle , , , , , 2, , , , , \rangle, \\ B &= \langle , , , , , 2, 3, , , \rangle, \\ B &= \langle , , , 1, , 2, 3, , , \rangle \end{aligned}$$

and at the end,

$$B = \langle 0, 0, 1, 1, 2, 2, 3, 3, 4, 6, 6 \rangle.$$

8.2-2

> Prove that COUNTING-SORT is stable.

Suppose positions i and j with $i < j$ both contain some element k. We consider lines 10 through 12 of COUNTING-SORT, where we construct the output array. Since $j > i$, the loop will examine $A[j]$ before examining $A[i]$. When it does so, the algorithm correctly places $A[j]$ in position $m = C[k]$ of B. Since $C[k]$ is decremented in line 12, and is never again incremented, we are guaranteed that when the **for** loop examines $A[i]$ we will have $C[k] < m$. Therefore $A[i]$ will be placed in an earlier position of the output array, proving stability.

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 algorithm still works correctly. The order that elements are taken out of C and put into B doesn't affect the placement of elements with the same key. It will still fill the interval $(C[k-1], C[k]]$ with elements of key k. The question of whether it is stable or not is not well phrased. In order for stability to make sense, we would need to be sorting items which have information other than their key, and the sort as written is just for integers, which don't. We could think of extending this algorithm by placing the elements of A into a collection of elements for each cell in array C. Then, if we use a FIFO collection, the modification of line 10 will make it stable, if we use LILO, it will be anti-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 $[a..b]$ in $O(1)$ time. Your algorithm should use $\Theta(n+k)$ preprocessing time.

The algorithm will begin by preprocessing exactly as COUNTING-SORT does in lines 1 through 9, so that $C[i]$ contains the number of elements less than or equal to i in the array. When queried about how many integers fall into a range $[a..b]$, simply compute $C[b] - C[a-1]$. This takes $O(1)$ times and yields the desired output.

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.

0	1	2	3
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	BI**G**	B**I**G	**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	**S**EA
TEA	NO**W**	B**O**X	**T**AB
NOW	BO**X**	F**O**X	**T**AR
FOX	FO**X**	R**U**G	**T**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 preprocess, replacing each element of an array with an ordered pair. The first entry will be the value of the element, and the second value will be the index of the element.

For example, the array $[2, 1, 1, 3, 4, 4, 4]$ would become $[(2, 1), (1, 2), (1, 3), (3, 4), (4, 5), (4, 6), (4, 7)]$. We now interpret $(i, j) < (k, m)$ if $i < k$ or $i = k$ and $j < m$. Under this definition of less-than, the algorithm is guaranteed to be stable because each of our new elements is distinct and the index comparison ensures that if a repeat element appeared later in the original array, it must appear later in the sorted array. This doubles the space requirement, but the running time will be asymptotically unchanged.

8.3-3

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

Loop invariant: At the beginning of the **for** loop, the array is sorted on the last $i - 1$ digits.

Initialization: The array is trivially sorted on the last 0 digits.

Maintenance: Let's assume that the array is sorted on the last $i-1$ digits. After we sort on the ith digit, the array will be sorted on the last i digits. It is obvious that elements with different digit in the ith position are ordered accordingly; in the case of the same ith digit, we still get a correct order, because we're using a stable sort and the elements were already sorted on the last $i-1$ digits.

Termination: The loop terminates when $i = d + 1$. Since the invariant holds, we have the numbers sorted on d digits.

8.3-4

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

First run through the list of integers and convert each one to base n, then radix sort them. Each number will have at most $\log_n n^3 = 3$ digits so there will only need to be 3 passes. For each pass, there are n possible values which can be taken on, so we can use counting sort to sort each digit in $O(n)$ time.

8.3-5 ⋆

> 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?

Given n d-digit numbers in which each digit can take on up to k possible values, we'll perform $\Theta(k^d)$ passes and keep track of $\Theta(nk)$ piles in the worst case.

8.4 Bucket sort

8.4-1

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

R	
0	
1	.13.16
2	.20
3	.39
4	.42
5	.53
6	.64
7	.79.71
8	.89
9	

$$A = \langle .13, .16, .20, .39, .42, .53, .64, .71, .79, .89 \rangle.$$

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 $O(n \lg n)$?

If all the keys fall in the same bucket and they happen to be in reverse order, we have to sort a single bucket with n items in reversed order with insertion sort. This is $\Theta(n^2)$.

We can use merge sort or heapsort to improve the worst-case running time. Insertion sort was chosen because it operates well on linked lists, which has optimal time and requires only constant extra space for short linked lists. If we use another sorting algorithm, we have to convert each list to an array, which might slow down the algorithm in practice.

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 $E[X^2]$? What is $E^2[X]$?

$$ \begin{aligned} ☑ & = 2 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1 \\ \text E[X^2] & = 4 \cdot \frac{1}{4} + 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} = 1.5 \\ ☑ & = \text E[X] \cdot \text E[X] = 1 \cdot 1 = 1. \end{aligned} $$

8.4-4 ⋆

> 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.)

Bucket sort by radius,

$$\pi r_i^2 = \frac{i}{n} \cdot \pi 1^2$$
$$r_i = \sqrt{\frac{i}{n}}.$$

8.4-5 ⋆

> 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.

Bucket sort by p_i, so we have n buckets: $[p_0, p_1), [p_1, p_2), \ldots, [p_{n-1}, p_n)$. Note that not all buckets are the same size, which is ok as to ensure linear run time, the inputs should on average be uniformly distributed amongst all buckets, of which the intervals defined with p_i will do so.

p_i is defined as follows:

$$P(p_i) = \frac{i}{n}.$$

Problem 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 $\text{RANDOM}(1, r)$ 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. There are $n!$ possible permutations of the input array because the input elements are all distinct. Since each is equally likely, the distribution is uniformly supported on this set. So, each occurs with probability $\frac{1}{n!}$ and corresponds to a different leaf because the program needs to be able to distinguish between them.

b. The depths of particular elements of LT or RT are all one less than their depths when considered elements of T. In particular, this is true for the leaves of the two subtrees. Also, $\{LT, RT\}$ form a partition of all the leaves of T. Therefore, if we let $L(T)$ denote the leaves of T,

$$\begin{aligned} D(T) &= \sum_{\ell \in L(T)} D_T(\ell) \\ &= \sum_{\ell \in L(LT)} D_T(\ell) + \sum_{\ell \in L(RT)} D_T(\ell) \\ &= \sum_{\ell \in L(LT)} (D_{LT}(\ell) + 1) + \sum_{\ell \in L(RT)} (D_{RT}(\ell) + 1) \\ &= \sum_{\ell \in L(LT)} D_{LT}(\ell) + \sum_{\ell \in L(RT)} D_{RT}(\ell) + k \\ &= D(LT) + D(RT) + k. \end{aligned}$$

c. Suppose we have a T with k leaves so that $D(T) = d(k)$. Let i_0 be the number of leaves in LT. Then, $d(k) = D(T) = D(LT) + D(RT) + k$. However, we can pick LT and RT to minimize the external path length.

d. We treat i as a continuous variable, and take a derivative to find critical points. The given expression has the following as a derivative with respect to i

$$\frac{1}{\ln 2} + \lg i + \frac{1}{\ln 2} - \lg(k - i) = \frac{2}{\ln 2} + \lg\left(\frac{i}{k - i}\right),$$

which is 0 when we have $\frac{i}{k-i} = 2^{-\frac{2}{\ln 2}} = 2^{-\lg e^2} = e^{-2}$. Therefore, $(1 + e^{-2})i = k$, $i = \frac{k}{1+e^{-2}}$.

Since we are picking the two subtrees to be roughly equal size, the total depth will be order $\lg k$, with each level contributing k, so the total external path length is at least $k \lg k$.

e. Since before we that a tree with k leaves needs to have external length $k \lg k$, and that a sorting tree needs at least $n!$ trees, a sorting tree must have external tree length at least $n! \lg(n!)$. Since the average case run time is the depth of a leaf weighted by the probability of that leaf being the one that occurs, we have that the run time is at least $\frac{n! \lg(n!)}{n!} = \lg(n!) \in \Omega(n \lg n)$.

f. Since the expected runtime is the average over all possible results from the random bits, if every possible fixing of the randomness resulted in a higher runtime, the average would have to be higher as well.

Problem 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 from 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. Counting-Sort.

b. Quicksort-Partition.

c. Insertion-Sort.

d. (a) Yes. (b) No. (c) No.

e.

Thanks @Gutdub for providing the solution in this issue.

```
MODIFIED-COUNTING-SORT(A, k)
    let C[0..k] be a new array
    for i = 1 to k
        C[i] = 0
    for j = 1 to A.length
        C[A[j]] = C[A[j]] + 1
    for i = 2 to k
        C[i] = C[i] + C[i - 1]
    insert sentinel element NIL at the start of A
    B = C[0..k - 1]
    insert number 1 at the start of B
    // B now contains the "endpoints" for C
    for i = 2 to A.length
        while C[A[i]] != B[A[i]]
            key = A[i]
            exchange A[C[A[i]]] with A[i]
            while A[C[key]] == key // make sure that elements with the same
keys will not be swapped
                C[key] = C[key] - 1
```

```
    remove the sentinel element
    return A
```

In place (storage space is $\Theta(k)$) but not stable.

Problem 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. First, sort the integer according to their lengths by bucket sort, where we make a bucket for each possible number of digits. We sort each these uniform length sets of integers using radix sort. Then, we just concatenate the sorted lists obtained from each bucket.

b. Make a bucket for every letter in the alphabet, each containing the words that start with that letter. Then, forget about the first letter of each of the words in the bucket, concatenate the empty word (if it's in this new set of words) with the result of recursing on these words of length one less. Since each word is processed a number of times equal to it's length, the runtime will be linear in the total number of letters.

Problem 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 make.

> **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. Select a red jug. Compare it to blue jugs until you find one which matches. Set that pair aside, and repeat for the next red jug. This will use at most $\sum_{i=1}^{n-1} i = n(n-1)/2 = \Theta(n^2)$ comparisons.

b. We can imagine first lining up the red jugs in some order. Then a solution to this problem becomes a permutation of the blue jugs such that the ith blue jug is the same size as the ith red jug. As in section 8.1, we can make a decision tree which represents comparisons made between blue jugs and red jugs. An internal node represents a comparison between a specific pair of red and blue jugs, and a leaf node represents a permutation of the blue jugs based on the results of the comparison. We are interested in when one jug is greater than, less than, or equal in size to another jug, so the tree should have 3 children per node. Since there must be at least $n!$ leaf nodes, the decision tree must have height at least $\log_3(n!)$. Since a solution corresponds to a simple path from root to leaf, an algorithm must make at least $\Theta(n \lg n)$ comparisons to reach any leaf.

c. We use an algorithm analogous to randomized quicksort. Select a blue jug at random. Partition the red jugs into those which are smaller than the blue jug, and those which are larger. At some point in the comparisons, you will find the red jug which is of equal size. Once the red jugs have been divided by size, use the red jug of equal size to partition the blue jugs into those which are smaller and those which are larger. If k red jugs are smaller than the originally chosen jug, we need to solve the original problem on input of size $k-1$ and size $n-k$, which we will do in the same manner. A subproblem of size 1 is trivially solved because if there is only one red jug and one blue jug, they must be the same size. The analysis of expected number of comparisons is exactly the same as that of RANDOMIZED-QUICKSORT given on pages 181-184. We are running the procedure twice so the expected number of comparisons is doubled, but this is absorbed by the big-O notation. In the worst case, we pick the largest jug each time, which results in $\sum_{i=2}^{n} i + i - 1 = n^2$ comparisons.

Problem 8-5 Average sorting

> 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:
>
> $$\frac{\sum_{j=i}^{i+k-1} A[j]}{k} \leq \frac{\sum_{j=i+1}^{i+k} A[j]}{k}.$$
>
> **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 $A[i] \leq A[i+k]$ 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. Ordinary sorting

b. $2, 1, 4, 3, 6, 5, 8, 7, 10, 9$.

c.

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

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

$$A[i] \leq A[i+k].$$

d. Shell-Sort, i.e., We split the n-element array into k part. For each part, we use Insertion-Sort (or Quicksort) to sort in $O(n/k \lg(n/k))$ time. Therefore, the total running time is $k \cdot O(n/k \lg(n/k)) = O(n \lg(n/k))$.

e. Using a heap, we can sort a k-sorted array of length n in $O(n \lg k)$ time. (The height of the heap is $\lg k$, the solution to Exercise 6.5-9.)

f. The lower bound of sorting each part is $\Omega(n/k \lg(n/k))$, so the total lower bound is $\Theta(n \lg(n/k))$. Since k is a constant, therefore $\Theta(n \lg(n/k)) = \Omega(n \lg n)$.

Problem 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 tighter $2n - 1$ 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. There are $\binom{2n}{n}$ ways to divide $2n$ numbers into two sorted lists, each with n numbers.

b. Based on Exercise C.1.13,

$$\begin{aligned}\binom{2n}{n} &\le 2^h \\ h &\ge \lg \frac{(2n)!}{(n!)^2} \\ &= \lg(2n!) - 2\lg(n!) \\ &= \Theta(2n \lg 2n) - 2\Theta(n \lg n) \\ &= \Theta(2n).\end{aligned}$$

c. We have to know the order of the two consecutive elements.

d. Let list $A = 1, 3, 5, \ldots, 2n - 1$ and $B = 2, 4, 6, \ldots, 2n$. By part (c), we must compare 1 with 2, 2 with 3, 3 with 4, and so on up until we compare $2n - 1$ with $2n$. This amounts to a total of $2n - 1$ comparisons.

Problem 8-7 The 0-1 sorting lemma and columnsort

> A ***compare-exchange*** operation on two array elements $A[i]$ and $A[j]$, where $i < j$, has the form
>
> ```
> COMPARE-EXCHANGE(A, i, j)
>     if A[i] > A[j]
>         exchange A[i] with A[j]
> ```
>
> After the compare-exchange operation, we know that $A[i] \le A[j]$.
>
> An ***oblivious compare-exchange algorithm*** operates solely by a sequence of prespecified compare-exchange operations. The indices of the positions compared in the sequence must be determined in advance, and although they can depend on the number of elements being sorted, they cannot depend on the values being sorted, nor can they depend on the result of any prior compare-exchange operation. For example, here is insertion sort expressed as an oblivious compare-exchange algorithm:
>
> ```
> INSERTION-SORT(A)
>     for j = 2 to A.length
>         for i = j - 1 downto 1
>             COMPARE-EXCHANGE(A, i, i + 1)
> ```

The ***0-1 sorting lemma*** provides a powerful way to prove that an oblivious compare-exchange algorithm produces a sorted result. It states that if an oblivious compare-exchange algorithm correctly sorts all input sequences consisting of only 0s and 1s, then it correctly sorts all inputs containing arbitrary values.

You will prove the 0-1 sorting lemma by proving its contrapositive: if an oblivious compare-exchange algorithm fails to sort an input containing arbitrary values, then it fails to sort some 0-1 input. Assume that an oblivious compare-exchange algorithm X fails to correctly sort the array $A[1..n]$. Let $A[p]$ be the smallest value in A that algorithm X puts into the wrong location, and let $A[q]$ be the value that algorithm X moves to the location into which $A[p]$ should have gone. Define an array $B[1..n]$ of 0s and 1s as follows:

$$B[i] = \begin{cases} 0 & \text{if } A[i] \le A[p], \\ 1 & \text{if } A[i] > A[p]. \end{cases}$$

a. Argue that $A[q] > A[p]$, so that $B[p] = 0$ and $B[q] = 1$.

b. To complete the proof of the 0-1 sorting lemma, prove that algorithm X fails to sort array B correctly.

Now you will use the 0-1 sorting lemma to prove that a particular sorting algorithm works correctly. The algorithm, ***columnsort***, works on a rectangular array of n elements. The array has r rows and s columns (so that $n = rs$), subject to three restrictions:

- r must be even,
- s must be a divisor of r, and
- $r \ge 2s^2$.

When columnsort completes, the array is sorted in ***column-major order***: reading down the columns, from left to right, the elements monotonically increase.

Columnsort operates in eight steps, regardless of the value of n. The odd steps are all the same: sort each column individually. Each even step is a fixed permutation. Here are the steps:

1. Sort each column.
2. Transpose the array, but reshape it back to r rows and s columns. In other words, turn the leftmost column into the top r/s rows, in order; turn the next column into the next r/s rows, in order; and so on.
3. Sort each column.
4. Perform the inverse of the permutation performed in step 2.
5. Sort each column.
6. Shift the top half of each column into the bottom half of the same column, and shift the bottom half of each column into the top half of the next column to the right. Leave the top half of the leftmost column empty. Shift the bottom half of the last column into the top half of a new rightmost column, and leave the bottom half of this new column empty.
7. Sort each column.
8. Perform the inverse of the permutation performed in step 6.

> Figure 8.5 shows an example of the steps of columnsort with $r = 6$ and $s = 3$. (Even though this example violates the requirement that $r \geq 2s^2$, it happens to work.)
>
> **c.** Argue that we can treat columnsort as an oblivious compare-exchange algorithm, even if we do not know what sorting method the odd steps use.
>
> Although it might seem hard to believe that columnsort actually sorts, you will use the 0-1 sorting lemma to prove that it does. The 0-1 sorting lemma applies because we can treat columnsort as an oblivious compare-exchange algorithm. A couple of definitions will help you apply the 0-1 sorting lemma. We say that an area of an array is ***clean*** if we know that it contains either all 0s or all 1s. Otherwise, the area might contain mixed 0s and 1s, and it is ***dirty***. From here on, assume that the input array contains only 0s and 1s, and that we can treat it as an array with r rows and s columns.
>
> **d.** Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most s dirty rows between them.
>
> **e.** Prove that after step 4, the array, read in column - major order, starts with a clean area of 0s, ends with a clean area of 1s, and has a dirty area of at most s^2 elements in the middle.
>
> **f.** Prove that steps 5–8 produce a fully sorted 0-1 output. Conclude that columnsort correctly sorts all inputs containing arbitrary values.
>
> **g.** Now suppose that s does not divide r. Prove that after steps 1–3, the array consists of some clean rows of 0s at the top, some clean rows of 1s at the bottom, and at most $2s - 1$ dirty rows between them. How large must r be, compared with s, for columnsort to correctly sort when s does not divide r?
>
> **h.** Suggest a simple change to step 1 that allows us to maintain the requirement that $r \geq 2s^2$ even when s does not divide r, and prove that with your change, columnsort correctly sorts.

(Removed)