3. Sorting and Algorithm Analysis

In elementary school, students begin their studies by learning the fundamental topics of reading, writing, and arithmetic. In the study of algorithms, students begin by learning to sort. Given n comparable elements (e.g., numbers or text strings), the sorting problem asks us to arrange them in nondecreasing order. Sorting is a truly fundamental algorithmic topic. It has been the focus of hundreds of publications in the computer science literature, and it is almost always the first substantial topic covered in any algorithms course. Vast amounts of computing power are spent solving sorting problems in practice, and sorting is also a key preprocessing step or subroutine for many more sophisticated algorithms.

This chapter serves not only as an introduction to sorting algorithms, but also as an introduction to fundamental techniques for designing algorithms and analyzing their correctness and running time. Sorting is the ideal domain for such a discussion, due to the wide variety of common sorting algorithms out there. We also study two problems closely related to sorting: *selection* (finding the kth largest element in an unsorted sequence), and *topological sorting* a partially-ordered sequence (where only some pairs of elements can be meaningfully compared).

We assume for simplicity that our input comes in an array $A[1 \ldots n]$, which we then want to permute so that $A[1] \leq A[2] \leq A[3] \leq \ldots \leq A[n]$. Many sorting algorithms can easily be adapted to work directly on linked lists with no degradation in running time. When sorting large records, it is often faster in practice to sort an array of pointers to these records, since this involves moving less memory around.

Most of the algorithms in this chapter are designed for the comparison-based model of computation, where we assume that input elements can be compared pairwise, and nothing more. If our input elements are numbers, we can also consider using the RAM or real RAM models. The real RAM turns to be not very different from the comparison-based model for sorting. We will soon show that there is an $\Omega(n \log n)$ worst-case lower bound for sorting in both of these models, so sorting algorithms running in $O(n \log n)$ time such as merge sort and quicksort are optimal in this setting. In the RAM model, since we can assume our input consists of integers, we will be able to design algorithms with input-sensitive running times that can conceivably run faster, as long as they are sorting integers that are sufficiently small. For example, *radix sort* only takes $\Theta(n)$ time to sort n integers of size at most n^c, where c is a constant.

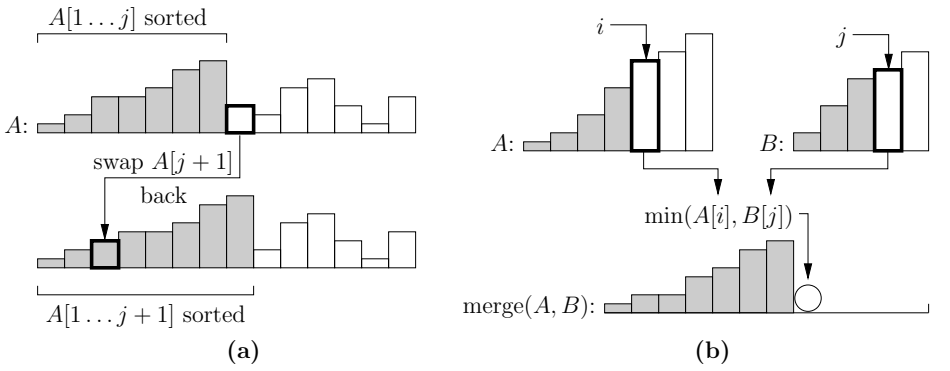

FIGURE 3.1: Illustrations of (a) a single iteration of insertion sort, where $A[j+1]$ is inserted within the sorted prefix $A[1 \ldots j]$, and (b) a snapshot in the middle of the process of merging two sorted arrays A and B.

3.1 Algorithm Design Techniques

Our discussion begins with three simple sorting algorithms that illustrate some of the most basic techniques of algorithm design. These will then serve as examples for how to argue both the correctness and running time of an algorithm.

3.1.1 Iterative Refinement: Bubble Sort

Iterative refinement involves starting with an arbitrary (probably incorrect) solution and repeatedly improving it with small modifications until it eventually becomes correct. Many prominent algorithms are based on this simple idea, including a wide range of heuristics for obtaining good solutions to hard problems (Chapter 13) and algorithms for solving optimization problems (Chapter 14).

At the heart of any iterative refinement algorithm is usually a subroutine that either proclaims a solution to be correct, or identifies some aspect of the solution that can be modified to make it more correct. For example, if we scan through an array A and realize it is not sorted, then there must be some adjacent out-of-order pair of elements $A[i] > A[i+1]$ that we can swap to make the array "more sorted" (we will formalize this notion in a few pages when we talk about inversions). This leads to the $O(n^2)$-time *bubble sort* algorithm, which repeatedly scans through our array, swapping any adjacent out-of-order pairs of elements it finds, stopping once the array becomes sorted. Its name reflects the way it causes small elements to slowly drift toward the front of the array just as bubbles drift to the top of a pool of water.

3.1.2 Incremental Construction: Insertion Sort

When sorting a stack of papers, you may instinctively use an *insertion sort*, where you maintain two stacks of papers: those sorted so far, and the remaining unsorted ones. In every step, you insert a paper from the unsorted stack into its proper

location in the sorted stack. Insertion sort is an example of an algorithm design technique called *incremental construction*, where a solution is built up step by step, one element at a time.

When applying insertion sort to an array $A[1 \ldots n]$, the "sorted stack" consists of some prefix $A[1 \ldots j]$ whose elements are in sorted order. We expand this prefix in each iteration, as shown in Figure 3.1(a), by moving the next element $A[j+1]$ into its proper location so as to leave $A[1 \ldots j+1]$ sorted. This is sometimes done by taking $A[j+1]$ and repeatedly swapping it backwards as long as it is preceded by a larger element. It takes $O(n)$ time to process each successive element, for a total running time of $O(n^2)$.

We can also think of incremental algorithms from a recursive, rather than iterative, point of view. For example, we could say that insertion sort first recursively sorts $A[1 \ldots n-1]$, then inserts the final element $A[n]$ into its proper location so the whole array becomes sorted. The difference is only in how we choose to think about the algorithm; both variants perform essentially the same operations and have identical running times.

3.1.3 Divide and Conquer: Merge Sort

Many algorithms construct a large solution out of recursively-computed solutions to smaller instances of the same problem. With incremental construction, we successively add one element at a time, taking a solution to a subproblem of size $n-1$ and somehow augmenting it with one additional element. By way of contrast, *divide and conquer* algorithms tend to decompose a larger problem in a more "multiplicative" fashion — for example, splitting a problem of size n into 2 subproblems of size $n/2$.

A popular sorting algorithm based on divide and conquer is *merge sort*, based on the fundamental process of *merging* two sorted sequences into one larger sorted sequence. Merging is quite straightforward — to merge two sorted stacks of paper, you can repeatedly compare the front pages of both stacks, always selecting the smaller for the next page in the merged stack. As shown in Figure 3.1(b), we merge two arrays A and B the same way. We maintain pointers i and j to the "front" elements of both arrays, and every iteration we select the smaller of these to be the next element in the merged array, advancing its corresponding pointer. Alternatively, from a recursive point of view, we select the smaller of $A[1]$ and $B[1]$ to be first in the merged array, and the rest of the merged array is formed by recursively merging the leftover parts of A and B. It takes $\Theta(n)$ time to merge two arrays of combined length n, since each step places one element into its correct final position in the merged array.

To merge sort an array A, we recursively merge sort the first half $A[1 \ldots n/2]$ and second half $A[n/2+1 \ldots n]$, then merge the results together. If you prefer an iterative outlook, the same process can be described as in Figure 3.2: regard an n-element array as n adjacent 1-element sorted arrays, then merge these pairwise to obtain $n/2$ adjacent 2-element sorted arrays, then $n/4$ adjacent 4-element sorted arrays, and so on. Both processes do the same work, and differ only in whether we prefer a "top down", recursive perspective (arguably the more natural way to describe a divide and conquer algorithm like merge sort) or a "bottom up", iterative perspective. Merge sort runs in $\Theta(n \log n)$ time, as we shall see in a moment.

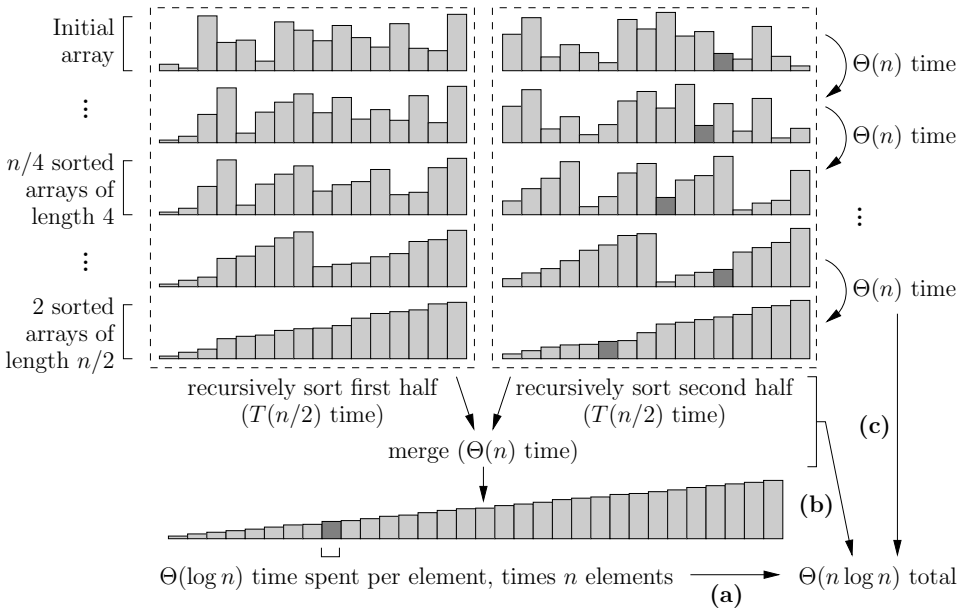

FIGURE 3.2: Running time of merge sort computed three ways: (a) by noting that we spend $\Theta(\log n)$ time per element, (b) by solving the recurrence $T(n) = 2T(n/2) + \Theta(n)$, and (c) via an iterative perspective where the algorithm runs in $\log n$ phases, each taking $\Theta(n)$ time.

3.2 Arguing Correctness and Running Time

Although it may seem obvious that the three sorting algorithms above are correct, caution is advised — algorithms that seem correct at first glance may often fail for subtle reasons, and only with a clear mathematical analysis can one establish correctness beyond any reasonable doubt. We can argue correctness at many levels of detail. In this book, we tend to stay at a high level, although for mission-critical applications one can also argue correctness of an algorithm or computer program in excruciating detail.

In this section, we discuss techniques for analyzing correctness and running time. Correctness is the most important of the two, of course, since there is still value in a correct algorithm whose running time we do not fully understand, but a fast incorrect algorithm is typically of little use. Showing that an algorithm eventually terminates is also a key part of proving correctness. For iterative refinement algorithms, termination usually implies that the algorithm has reached a correct solution, so the entire correctness argument reduces to proving termination.

3.2.1 Correctness Proofs Based on Induction

Many correctness arguments are structured as proofs by induction. This is particularly well suited for algorithms that build large solutions out of recursively-

constructed smaller solutions, since it allows us to assume by induction on problem size that our algorithm will correctly solve any subproblem of strictly smaller size. Consider merge sort for example:

Claim: *Merge sort correctly sorts any array of length n.*

Proof: This is easily proved using induction on n. As a base case (a key part of any inductive proof!), correctness is trivially verified for $n = 1$. For $n > 1$, induction tells us that our recursive calls to merge sort will properly sort the first and second halves of the array. All that remains is to argue correctness of our merging algorithm, which we also accomplish using induction. Exactly how we apply induction in this case, however, depends on whether we are phrasing the merge as an iterative (i.e., loop-based) algorithm or as a recursive algorithm. Recall that a simple recursive way to merge two sorted lists A and B is to take the smaller of their initial elements as the first element in the merged list, and then to complete the output by recursively merging the left-over contents of A and B. Here, correctness easily follows from induction on the combined length of A and B: the first element we choose is clearly the smallest overall, and hence the correct element to place first in the merged list. We then claim by induction that the remainder of A and B will be merged correctly, since this constitutes a smaller problem instance.

Iterative Algorithms and Loop Invariants. Induction proofs are also often applied to iterative algorithms, Here, we usually apply induction on the number of iterations of our algorithm, and our inductive hypothesis is called a *loop invariant*. An *invariant* is a property that always holds at key points during an algorithm's execution, and a loop invariant is a condition that remains true at the beginning of every iteration of a loop. As a simple example, consider randomly permuting the contents of an array $A[1 \ldots n]$ as follows:

```
For j = 1...n:
  Let i be a random number in {1,...,j}
  Swap A[i] and A[j]
```

A suitable loop invariant here is that the elements in $A[1 \ldots j-1]$ are equally likely to appear in any of their $(j-1)!$ possible permutations. This is true as a base case when the loop begins, and one can show that it is maintained by each iteration of the loop [Easy proof]. Upon termination, when j reaches $n+1$, the invariant implies that we have indeed randomly permuted A's contents.

Problem 39 (Correctness of Iterative Merging). Using an appropriate loop invariant, please give a short proof of correctness for the iterative approach for merging shown in Figure 3.1(b). [Solution]

3.2.2 Different Ways of Adding up the Running Time

There are several methods we can use to determine the total running time of an algorithm, depending on its structure.

Solving a Recurrence. For recursive algorithms, we usually compute running time by solving a recurrence. For example, if $T(n)$ denotes the running time of merge sort on n elements, we know that $T(n) = 2T(n/2) + \Theta(n)$, since we are performing two recursive sorts each on $n/2$ elements, followed by a $\Theta(n)$ merge

operation. The solution of this recurrence gives the running time of merge sort, $T(n) = \Theta(n \log n)$.

Loop Counting. For iterative algorithms, simple loop counting often suffices. As shown in Figure 3.2, we can think of merge sort as an iterative algorithm that takes n singleton elements and merges them pairwise to obtain $n/2$ sorted lists of length 2, then merges these pairwise to obtain $n/4$ sorted lists of length 4, and so on for $\log n$ such phases. Since merging is a linear-time operation, each phase (merging n/k sorted lists of length k) takes $\Theta(n)$ time, for a total of $\Theta(n \log n)$ time[1].

Running Time Spent per Element. Instead of adding up the total time spent in each step of an algorithm, summed over all steps during the algorithm's execution, it is sometimes more convenient to consider the total running time spent on each individual input element, summed over all elements. This adds up the same amount of total work, just in a different, more convenient order. Using merge sort as our example again (Figure 3.2), it takes $\Theta(n)$ time to merge two sorted arrays of combined length n, which is $O(1)$ time per element taking part in the merge. The total amount of work merge sort spends on a single element of data is therefore proportional to the number of merge operations in which the element takes part. How many merges happen to a single element? If you put yourself in the perspective of an element of data, you will find that every time you take part in a merge, you end up in a sorted subarray twice as large as before. This will happen $\log n$ times before you end up in a sorted array of length n. We therefore spend $\Theta(\log n)$ time per element, for a total running time of $\Theta(n \log n)$. This sort of "per element" outlook on running time can be highly useful in the analysis of many algorithms.

3.2.3 Inversions and Potential Arguments

Another common and powerful technique for proving correctness and running time is the use of a *potential function* argument. A potential function maps the state of our algorithm to a nonnegative integer. If we can guarantee that this must decrease during each iteration of the algorithm, then termination is inevitable. This approach is commonly used with algorithms based on iterative refinement, where our potential function usually reflects the amount of "incorrectness" inherent in the algorithm's current solution. There is a natural physical analogy suggested by the use of the term "potential": we can think of a potential function as telling us the amount of "energy" stored in our current state, with our algorithm acting in the role of gravity, pulling in a direction that decreases this potential energy.

For sorting, a natural potential function is the number of *inversions* in our array. A pair of elements $A[i]$ and $A[j]$ with $i < j$ constitutes an inversion if $A[i] > A[j]$; that is, the elements are ordered incorrectly with respect to their positions. A sorted array has no inversions, and a reverse-sorted array has the maximum possible number of inversions, $\binom{n}{2}$, since every pair is an inversion. Inversions show up often in the study of permutations and sorting, and since inversion count is a natural way to measure "unsortedness", it is often a good choice for a potential function. By

[1] Recall from Section 2.3 (Figure 2.5) that solving a recurrence involves adding up the work done by an algorithm at each "level" of recursion. If you look at Figure 3.2 closely, you will also see a tree of recursive subproblems whose work we have added up level by level. Hence, in this case, the analysis of our iterative outlook on merge sort turns out to be just another way of doing the same math we used to solve the merge sort recurrence.

swapping two adjacent out-of-order elements $A[i] > A[i+1]$, we correct a single inversion, leaving all others unchanged. For example, this tells us that bubble sort must terminate, since each scan through the array (except the last) performs at least one such swap, decreasing the inversion count.

Potential functions are often useful for analyzing running time. For example:

- **Bubble Sort.** Let A' denote the sorted version of our array A, and suppose our potential function tells us the size of the largest suffix of our array such that $A[j \dots n] = A'[j \dots n]$ (i.e., the number of elements at the end of A that are in their correct final positions). Each scan of bubble sort increases this potential by at least one. The first scan pulls the largest element, through repeated swaps, to its correct final position at the end of the array, then the second scan pulls the second-largest element back to the second-to-last position, and so on. Since our potential cannot exceed n, we perform at most n scans, each taking $\Theta(n)$ time, for a total running time of $O(n^2)$.

- **Insertion Sort.** Consider inversion count as a potential function. Since insertion sort has a fixed $\Theta(n)$ overhead for scanning the array and then corrects one inversion for each backward swap it makes, the running time is $\Theta(n + I)$ time, where I is the number of inversions in our input array. This is still $\Theta(n^2)$ in the worst case, but can potentially be much faster on nearly-sorted arrays[2], where it can even outperform merge sort. For example, if $I = \Theta(n)$, then insertion sort runs in $\Theta(n)$ time, while merge sort runs in $\Theta(n \log n)$ time (note that merge sort always runs in $\Theta(n \log n)$ time, even when given an already-sorted input).

When we study amortized analysis in the next chapter, we will make somewhat more sophisticated running time arguments by using potential functions that can both increase and decrease, rather than moving in a single monotonic direction.

Problem 40 (Inversions and Invariants). Suppose we wish to design a sorting algorithm in which the only operation available is a cyclic shift of 3 consecutive elements. For example, a left cyclic shift on the middle 3 characters of the string 'SORTING' yields 'SOTIRNG'. Using inversions and invariants, please characterize the inputs for which sorting is possible in this setting, and show how to sort them in $O(n^2)$ time. [Solution]

3.3 Lower Bounds

Bubble sort, insertion sort, and merge sort are all comparison-based algorithms. The comparison model is ideal for sorting, allowing us to develop general-purpose algorithms that can sort any type of comparable data (e.g., integers, real numbers, text strings, etc.). However, this generality has a price, as there is an $\Omega(n \log n)$ lower bound on the worst-case running time of any comparison-based algorithm for sorting n elements.

[2]Later in problem 105, we will design an "adaptive" variant of insertion sort whose running time scales gracefully as a function of I between $\Theta(n)$ for a sorted array (with no inversions) up to $\Theta(n \log n)$ for a reverse-sorted array (with the maximum possible number of inversions).

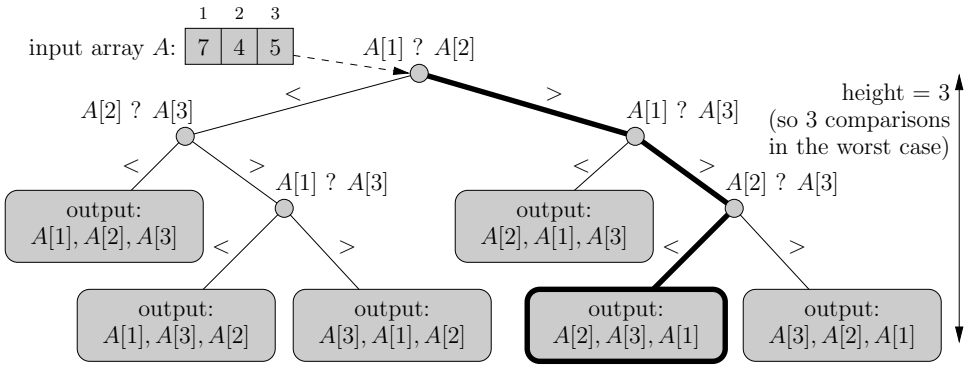

FIGURE 3.3: An comparison-based algorithm for sorting a 3-element array modeled as a decision tree. Elements are assumed to be distinct for simplicity, otherwise each comparison would generate three possible outputs: less than, greater than, or equal.

3.3.1 Modeling Algorithms by Decision Trees

We can easily show an upper bound of $O(n \log n)$ on the running time required to sort by demonstrating an $O(n \log n)$ algorithm, like merge sort. Arguing a lower bound is trickier, however, since this involves proving that *any* comparison-based sorting algorithm, no matter how bizarre or clever, must take $\Omega(n \log n)$ steps in the worst case. To make such a claim, we first show how any comparison-based sorting algorithm can be abstractly represented using a simple algorithmic model called a *decision tree*, then we show that any decision tree must make $\Omega(n \log n)$ comparisons in the worst case in order to properly sort n elements.

A sorting algorithm in the form of a decision tree is shown in Figure 3.3. At every step, it compares two elements and branches based on the result. The more comparisons the algorithm makes, the more it learns about the ordering of the input elements. When we reach a leaf, the algorithm terminates and declares how to rearrange the input in sorted order. The height of the tree is the number of comparisons required in the worst case. Any *deterministic* (not randomized) comparison-based sorting algorithm can be abstractly represented this way, as a series of decision trees, one for each input size n.

If we are sorting n distinct input elements, there are $n!$ different orderings we might receive as input. Each one needs to be re-arranged differently in order to sort them all properly. Our decision tree therefore must have at least $n!$ leaves. Otherwise, two different input orderings would find their way to the same leaf[3], and one of them would therefore by sorted incorrectly, since the same permutation would be applied to both. Since a decision tree of height h can have at most 2^h leaves, we need $2^h \geq n!$, so $h \geq \log n! = \Theta(n \log n)$ by Stirling's approximation. The height of our decision tree (i.e., the number of comparisons needed to sort n elements in the

[3]This is an example of the so-called *pigeonhole principle*: if there are more pigeons than pigeonholes, then two pigeons must end up in the same pigeonhole. This obvious yet important concept is used in many combinatorial proofs.

worst case) must therefore be $\Omega(n \log n)$.

Problems Involving Equality Testing. The result of a comparison is one of three outcomes: $=$, $<$, or $>$. Since any sorting algorithm must still properly sort in the special case where we have distinct elements, it suffices to consider only the cases $<$ and $>$ for the argument above. Equality plays a more central role in several other problems, however. For example:

- **Element Uniqueness**. Given n elements, the *element uniqueness problem* asks if they are all distinct or if there exist two equal elements.
- **Set Disjointness**. Given two sets A and B each with at most n elements, the *set disjointness problem* asks whether A and B share any elements in common (i.e., we want to test if $A \cap B = \emptyset$).
- **Set Equality**. Given two sets A and B with $|A| = |B| = n$, the *set equality problem* involves testing whether $A = B$.

For each of these, we can still use decision trees to argue an $\Omega(n \log n)$ worst-case lower bound in the comparison model, although they seem to require slightly more nuanced proofs[4] than the one above for sorting [Details]. Interestingly, we will see in Chapter 7 how to use "hashing" to solve all three problems in $\Theta(n)$ expected time in the RAM model, illustrating well the importance of our underlying computational model.

Problem 41 (Lower Bounds for Group Testing). Suppose up to k out of n individuals have a particular disease (typically with k much smaller than n). You would like to identify them with a minimum number of expensive tests. Of course, you could test all n individuals separately, but it can be much less costly to test groups of individuals. If we group together, say, blood samples from a set S of individuals, the outcome of a single test tells us whether (i) at least one individual in S has the disease, or (ii) nobody in S has the disease. You may have seen problems of this sort in recreational mathematics involving the identification of coins of incorrect weight by weighing groups of coins on a balance. Your solution can be "adaptive" in that results from one aggregate test can influence your choice of which group to test next (we consider the more challenging non-adaptive case later in problem 118). Please show a simple algorithm that identifies all of the sick individuals with $O(k \log n)$ tests, and use a decision tree argument to show that $\Omega(k \log \frac{n}{k})$ tests are necessary in the worst case. [Solution]

3.3.2 Randomized and Algebraic Decision Trees

To extend our $\Omega(n \log n)$ worst-case lower bound to *randomized* comparison-based sorting algorithms, let us think of a randomized algorithm abstractly as just a probability distribution over deterministic algorithms. That is, imagine taking all the random bits our algorithm will consume during its lifetime, and using these up front to select a deterministic algorithm from a huge table. After this initial selection, the algorithm is purely deterministic, since all of its formerly-random behavior has been fully specified. Owing to this insight, we can model a randomized

[4] We find it convenient to use concepts from topological sorting in our proofs here, so you may want to read ahead to Section 3.8 before listening to the proof details.

n-element comparison-based sorting algorithm as a probability distribution over decision trees. Since each of these trees must still sort properly (i.e., it must have $\Omega(n \log n)$ height), our $\Omega(n \log n)$ worst-case bound on comparisons still applies.

Of course, we rarely apply worst-case analysis to a randomized algorithm; it is far more typical to look at *expected* running time. Later in Section 12.5.4, we will see how to use techniques from game theory to prove lower bounds on the expected performance of randomized algorithms, and we will establish an $\Omega(n \log n)$ lower bound on the expected running time of any randomized n-element comparison-based sorting algorithm. This bound is closely related to the "average case" result that any deterministic comparison-based algorithm must take $\Omega(n \log n)$ expected time to sort n elements that have been permuted at random. Showing this is straightforward, by modifying the proof above to show that a decision tree with $n!$ leaves must have $\Omega(n \log n)$ *average* leaf depth.

It becomes more challenging to argue lower bounds when we move away from the comparison model towards a more general model like the real RAM, where input data is now numeric. For example, if we consider the element uniqueness problem defined above on a set of n real numbers $A[1 \ldots n]$, we could conceivably solve this problem by evaluating the product of $A[i] - A[j]$ over all pairs (i, j) and making only a single comparison to test if this product is equal to zero. While it may seem unlikely that this approach could possibly yield an efficient solution[5], the point is that it no longer suffices to consider only comparisons in order to obtain a meaningful lower bound.

To help us obtain lower bounds in the real RAM model, we turn to a generalization of the decision tree known as the *algebraic computation tree*, which models any real RAM algorithm by a tree containing both branching nodes at which we perform comparisons, and non-branching nodes at which we perform elementary operations such as addition, multiplication, division, etc. Further discussion of this model is beyond the scope of this book. However, one can show that sorting, as well as the other three example problems above (element uniqueness, set disjointness, and set equality) all have $\Omega(n \log n)$ worst-case lower bounds in the real RAM model through the use of algebraic computation trees.

3.3.3 Lower Bounds via Reductions

Now that we have non-trivial lower bounds for a small handful of problems, we can transfer these to other problems using *reductions*, somewhat like we used polynomial-time reductions to transfer hardness results from one problem to another in Section 1.6.3.

As an example, consider the problem of rearranging an array to group equal elements into contiguous blocks (not necessarily appearing in sorted order). An instance of the element uniqueness problem can be reduced to this problem, by grouping equal elements and then performing a $\Theta(n)$ scan to test whether each element is distinct from its neighbors. If we could group equal elements in time faster than $O(n \log n)$, the same would therefore be true for element uniqueness, contradicting its $\Omega(n \log n)$ worst-case lower bound in the comparison and real RAM models. Hence, it must

[5]Surprisingly, even though it seems to involve $\Theta(n^2)$ terms, this product can be evaluated in only $\Theta(n \log^2 n)$ time using the Fast Fourier Transform, as we shall see in Chapter **??**.

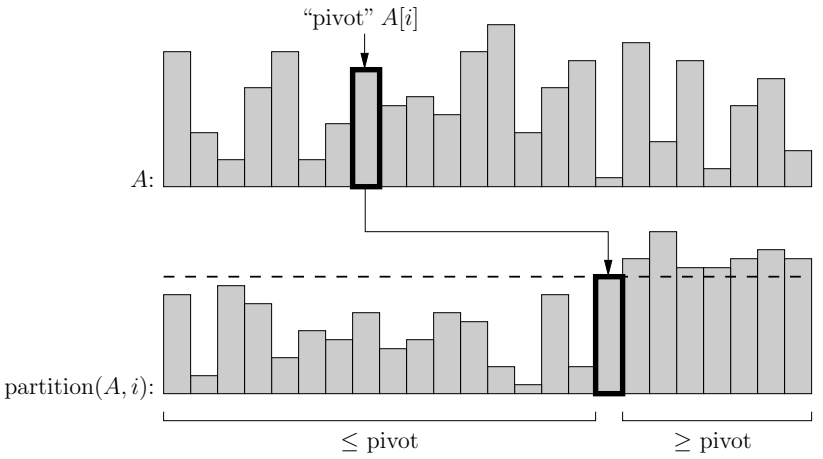

FIGURE 3.4: Illustration of the result of partitioning an array $A[1 \ldots n]$ about a pivot element $A[i]$. Although have drawn the output of the partition separate from the input, note that partitioning is usually done "in place" in the same memory space as the original array $A[1 \ldots n]$.

also take $\Omega(n \log n)$ time in these models to group equal elements.

Problem 42 (Reduction Practice). Please use reductions from sorting or other problems above (element uniqueness, set disjointness, set equality) to show $\Omega(n \log n)$ worst-case bounds for the following problems.

(a) Consider the following problems in the comparison model: (i) counting the number of occurrences of a most-frequently-occurring element in an n-element array, and (ii) given two sets A and B with $|A|+|B| = n$, compute $A \cap B$, $A \cup B$, or $A - B$. [Solution]

(b) Many problems in computational geometry inherit lower bounds from sorting, element uniqueness, or their relatives. Consider the following problems in the real RAM model, since they involve points with numeric coordinates: (i) the *closest pair* problem asks us to find the closest pair of points within a set of n points in the plane, and (ii) the *congruence testing* problem gives us two n-point sets A and B in the 2D plane, and asks whether B can be obtained from A by applying a rigid transformation — a translation plus a rotation plus (possibly) a reflection. [Solution]

3.4 Quicksort

One of the most popular and widely-used sorting algorithms is *quicksort*, which like merge sort is a based on the principle of divide and conquer. In fact, quicksort is almost symmetric to merge sort in its operation: merge sort performs two recursive sorts followed by a linear-time merge, whereas quicksort performs a linear-time *partition* followed by two recursive sorts. Partitioning is shown in Figure 3.4. Given a specific array element (known as a *pivot*), we rearrange the array into a block of elements less than or equal to the pivot, followed by the pivot element, followed by a block of elements greater than or equal to the pivot. The pivot element will

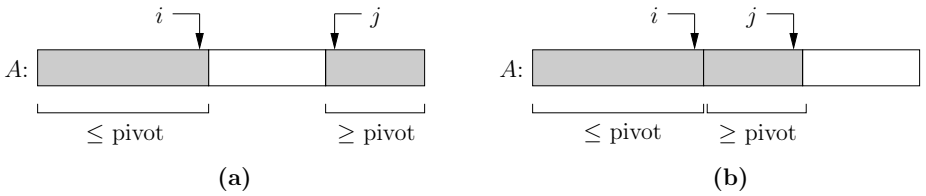

FIGURE 3.5: Two methods for partitioning an array in place. In (a), we scan pointers i and j inward from the ends of the array, stopping when $A[i]$ is larger than the pivot and $A[j]$ is smaller than the pivot. We then swap $A[i]$ and $A[j]$ and continue scanning inward, until the pointers meet. in (b), we scan two pointers from left to right. In each step, if $A[j+1]$ is larger than the pivot, we simply advance j; otherwise, we swap $A[i+1]$ and $A[j+1]$ and advance both i and j.

then be in its correct final location, and quicksort finishes by recursively sorting the subarrays left and right of the pivot.

Partitioning is very straightforward if we allocate a temporary buffer the same size as our array to hold the output. In this case, we simply scan through our array and copy out all the elements less than the pivot to the beginning of the buffer, and all those greater than the pivot to the end of the buffer. However, partitioning is more commonly performed "in place" without the need for an auxiliary buffer. Two common approaches for doing this are explained in Figure 3.5.

Choosing a Good Pivot. The tricky aspect of quicksort is how to choose the pivot. Ideally, we should choose the median element, which we will shortly learn how to find in only $\Theta(n)$ time. Since the median evenly splits our array into two recursive subproblems of size $n/2$, the running time of quicksort is then described by the recurrence $T(n) = 2T(n/2) + \Theta(n)$, with the $\Theta(n)$ term representing both the time spent finding the median and the time spent partitioning. In total, this solves to $T(n) = \Theta(n \log n)$. Unfortunately, the $\Theta(n)$ running time required to find the median element has a rather high hidden constant, so this version of quicksort is likely to run significantly slower than merge sort in practice, even though the two share the same asymptotic running time.

Our partition doesn't need to be perfectly balanced. Even if we could only guarantee at least a $1\% - 99\%$ split we would still end up with a running time of $\Theta(n \log n)$, albeit with a much higher hidden constant (since the solution of $T(n) = T(n/100) + T(99n/100) + \Theta(n)$ is still $\Theta(n \log n)$). On the other hand, if we always pick a very bad pivot like the minimum or maximum element, the running time can be as slow as $\Theta(n^2)$, since each $\Theta(n)$ partition operation makes very little progress, leaving us with a subproblem of size $n - 1$.

A simple but flawed strategy for selecting a pivot element is just to pick whichever element happens to be at the beginning or end of the array. Unfortunately, if our array is already sorted (a common occurrence in practice), this is a very bad choice. A slightly more popular heuristic is to look at the elements that appear first, last, and in the middle of the array and to use the median of these three as a pivot. This "median of three" approach can still lead to a running time of $\Theta(n^2)$ on contrived inputs, but in practice it has been observed to perform reasonably well.

Randomized Quicksort. Suppose we pick the pivot element at random[6], an approach that seems intuitively sensible since it should generate reasonable partitions most of the time. Indeed, this variant, known as *randomized quicksort*, runs in $\Theta(n \log n)$ time both in expectation and with high probability. Randomized quicksort is simple to implement and competitive with merge sort in practice.

To show that the expected running time of randomized quicksort is $\Theta(n \log n)$, we use linearity of expectation to decompose its running time into a sum of much simpler random variables. There are several nice ways to do this:

- We can add up the total number of comparisons performed by the algorithm (which is asymptotically the same as its running time), by defining an indicator random variable for each pair of elements taking the value 1 if they are compared, and 0 otherwise. [Details]
- We can add up the total expected work spent on subproblems of different sizes. That is, we take the sum, over all $k = 1 \dots n$, of the expected work spent partitioning all subproblems of size k seen during execution, the expected number of which is surprisingly easy to compute. [Details]
- We can show that $O(\log n)$ expected time is spent on each individual element in our array, using the randomized reduction lemma. [Details]

For a high probability bound, we observe from the last approach that the randomized reduction lemma also tells us that we spend $O(\log n)$ time on each element with high probability. By taking a union bound over all elements, the total running time is therefore also $O(n \log n)$ with high probability.

Problem 43 (Matching Nuts and Bolts). We are given n nuts of different sizes and n corresponding bolts, and we must match these together, determining for each nut the unique bolt of the same size. However, it is difficult to compare two nuts or two bolts — it is only possible to compare a nut against a bolt to see if the nut fits the bolt, if it is too small, or if it is too large. Describe a randomized algorithm running in $O(n \log n)$ time with high probability that will properly match the nuts and bolts. [Solution]

Stopping Early. Quicksort spends a lot of time on recursive function call overhead for very small subproblems. Therefore, a common trick to improve performance in practice is to "bottom out" of its recursion earlier by leaving sufficiently small arrays (say, with at most 4 elements) unsorted. This gives a final array that is nearly sorted, to which we apply insertion sort as a postprocessing step.

Quicksort and Binary Search Trees. In Chapter 6, we will learn about a powerful and versatile data structure called a *binary search tree*, which essentially encodes the recursive tree of all subproblems generated by quicksort. By saving this information, we can "dynamize" the sorting process, quickly computing changes to a sorted ordering in response to insertion or deletion of elements. Many fundamental algorithms related to sorting, such as binary search, quicksort, and quickselect (discussed shortly) are elegantly expressed in the context of binary search trees; we postpone further discussion until Chapter 6.

[6] Or, equivalently, we can use a simple deterministic pivoting strategy such as choosing the first element in the array, after randomly permuting the array as a preprocessing step.

3.5 In-Place Sorting

One advantage of quicksort over merge sort is the fact that quicksort can run *in place*. That is, it rearranges the contents of the input array using only the memory space allocated for the array, and almost no auxiliary storage. By contrast, merging n elements seems to require that we allocate a temporary array of size n to hold the output of the merge.

In modern times when computer memory is relatively cheap, there tends to be less emphasis on the importance of space versus running time. Indeed, time rather than space is typically the limiting resource for most algorithms in practice. However, for the case of sorting, memory usage tends to be studied a bit more closely, since many applications involve sorting very large data sets.

The strictest definition of an "in-place" algorithm would allow $O(1)$ extra words of memory in addition to the memory occupied by the input. However, this can make the implementation of recursive algorithms difficult, since there is often not enough stack space to store the state of unfinished recursive subproblems. Therefore, we typically call an algorithm "in-place" if it uses only $O(\log n)$ extra words of memory. Our earlier analysis of randomized quicksort showed that its stack depth is $O(\log n)$ with high probability, although the worst case is still $\Theta(n)$ if we have phenomenally bad luck in our selection of pivot elements. If we want to ensure $O(\log n)$ stack depth in the worst case irrespective of pivot quality, a clever trick is to have quicksort always recurse on the smaller of its two subproblems first. Since each such recursive call halves the size of our current subproblem, this limits recursion depth to $O(\log n)$. The second recursive call in quicksort is what is known as *tail recursion*. Since it is the last thing done by the quicksort function, there is no need for additional stack space to enact this call, since we don't need to save the state of the current function invocation. We simply re-use the same stack frame from the current invocation, effectively jumping right back to the start of the quicksort function with new parameters.

Problem 44 (In-Place Merge Sort). Standard merge sort does not operate in place, since it merges two arrays of combined length n into a newly-allocated memory buffer of length n. In this problem, your goal is to develop an *in-place* $O(n \log n)$ sorting algorithm based fundamentally on merging (it is of course easy to achieve this goal using other methods like quicksort). As a hint, your algorithm may be need to be somewhat "asymmetric", either by splitting into subproblems that are different in size, or by performing different tasks on different subproblems. As another hint, you can use the standard merging procedure with yet-unsorted parts of an array serving as scratch space. For example, if A, B, C, and D denote four quarters of an array, you can merge A and B together into the space occupied by C and D, while the elements of C and D become scrambled as they are swapped back into the space formerly occupied by A and B. [Solution]

3.6 Stable Sorting

A sorting algorithm is *stable* if it leaves equal elements in the same relative order as they were prior to the sort. This is useful when sorting large multi-field records. For example, consider the email messages in your inbox, each with a "from" address, a

date, and a subject line. If you sort according to date, and then stably by "from" address, then messages with the same "from" address would remain sorted by date.

Bubble sort, insertion sort, and merge sort are all stable, provided we implement them carefully. With quicksort, all methods we know for performing fast (linear time) in-place partitioning seem inherently unstable, so stability seems hard to achieve without sacrificing in-place operation. Stability and in-place operation seem somewhat at odds with each-other when it comes to fast sorting algorithms: merge sort is stable but not in-place, and quicksort, in-place merge sort (Problem 44), and heap sort (introduced in Chapter 5) are in-place but not stable. The question of whether stable and in-place sorting is possible in the comparison-based model in $O(n \log n)$ time remained open until the late 1970s, when algorithms fulfilling all of these requirements were finally discovered. However, to this day, all such algorithms remain extremely complicated. By contrast, as we see in the next problem, we can sort stably and in place in a very simple and elegant fashion if we are willing to settle for an $O(n \log^2 n)$ running time.

Problem 45 (Stable and In-Place Sorting). In this problem we construct variants of merge sort and quicksort that are stable, in place, and also reasonably efficient. These are the author's favorite sorting algorithms since they provide a very elegant demonstration of the power of the divide and conquer principle.

(a) To begin with, consider the problem of swapping two adjacent blocks in memory in place. That is, we have an array $A[1 \ldots n]$ in which we identify a left block $L = A[1 \ldots k]$ and a right block $R = A[k+1 \ldots n]$, and we would like to rearrange the contents of the array from LR to RL, with minimal additional memory usage. How can we do this in $\Theta(n)$ time? Note that L and R do not necessarily have the same size. [Solution]

(b) Using divide and conquer, design an $O(n \log n)$ stable in-place algorithm for merging. Building on this, we obtain a $O(n \log^2 n)$ stable in-place version of merge sort. The result from (a) may be of help. [Solution]

(c) Using divide and conquer, design an $O(n \log n)$ stable in-place algorithm for partitioning. Building on this, we obtain a $O(n \log^2 n)$ stable in-place version of randomized quicksort. The result from (a) may again be of help. [Solution]

(d) Using the $O(n \log n)$ stable in-place merging algorithm from part (b), give a simple alternative solution to problem 44. As a hint, start by dividing your array in half. [Solution]

Any sorting algorithm can be made stable by using additional memory (thus sacrificing in-place operation). To do this, we augment each element with its initial index within the array and modify our comparison operator to break ties between equal elements using these indices.

3.7 Selection

Selection is a close relative of the sorting problem, asking us to find the kth largest element (some might say kth smallest) in an unordered array $A[1 \ldots n]$. This element is said to have *rank* k, and is also called the kth *order statistic* of the sequence.

Examples include $k = 1$ (the minimum), $k = n$ (the maximum), and $k = n/2$ (the median, although if n is even, there is no unique median, and elements of ranks $k = n/2$ and $k = n/2 + 1$ could both rightfully be called "medians").

Selection is easy to solve in $O(n \log n)$ time by first sorting our array and then outputting $A[k]$. For the special cases $k = 1$ or $k = n$, it is easy to perform selection in $\Theta(n)$ time by simply scanning the array while keeping track of the smallest (or largest) element we see. Selection at other ranks is less trivial, although we shall see in this section how to select for the element of any rank k in only $\Theta(n)$ time. The median element is a particularly useful case, since it is commonly used as a splitting point in divide-and-conquer algorithms like quicksort to break a large problem two equal pieces.

Quickselect. We can solve the selection problem for any k in $\Theta(n)$ expected time using *quickselect*, a close relative of randomized quicksort. We first partition A about a randomly-chosen pivot element[7], after which we know the rank r of the pivot, since the pivot ends up in its correct final location if the array were sorted. If we are lucky enough that $k = r$, the pivot is the answer and we are done. Otherwise, by comparing k with r we can determine whether the element we seek lies on the left or right side of the pivot: if $k < r$, we recursively select for the element of rank k from subarray $A[1 \ldots r-1]$, and if $k > r$ we recursively select for the element of rank $k - r$ in the subarray $A[r+1 \ldots n]$. Quickselect is nearly identical in structure to randomized quicksort, except we recurse on only one of the two subarrays created by the partition operation, not both. This difference is sufficient to give only a $\Theta(n)$ expected running time[8]. [Proof]

Deterministic Selection. Using quickselect in conjunction with a clever but somewhat complicated approach for choosing suitable pivots deterministically, we can also solve the selection problem for any k in $\Theta(n)$ worst-case time. This is a nice result in theory, but since the resulting algorithm is rather complicated and also involves a large hidden constant, quickselect is usually the preferred method in practice. [How to perform deterministic selection in linear time]

Problem 46 (Selection of Multiple Order Statistics). The traditional selection problem involves computing one order statistic in a length-n array. Suppose instead that we have a list of k order statistics $s_1 \leq s_2 \leq \ldots \leq s_k$ that we would like to find. We can clearly compute all of these in $O(nk)$ time by applying a linear-time selection algorithm for each one individually. However, please show how to select for all k order statistics in only $O(n \log k)$ time. As an extreme case, if we set $s_j = j$ for $j = 1 \ldots k = n$, this boils down to essentially sorting, in $O(n \log n)$ time. [Solution]

Problem 47 (In-Place Selection from Read-Only Memory). We wish to perform selection from an array stored in read-only memory in an in-place setting. Rearranging the array elements is therefore prohibited, as is using more than $O(\log n)$ auxiliary storage. Give a randomized algorithm for this problem that runs in $O(n \log n)$ time with high probability. Interestingly, we do not know how to achieve a similarly-fast running time with a purely deterministic algorithm. [Solution]

[7] It is important to note that partitioning on the median is not an option here (as with quicksort), since finding the median requires selection, the problem we are currently trying to solve!

[8] This is one of the few randomized algorithms we will study where an expected running time analysis gives a stronger bound than a high probability analysis. If we want a high probability result for quickselect, $O(n \log n)$ is actually the best bound we can claim (and it is trivial to show this, since the algorithm is performing a subset of the work of randomized quicksort).

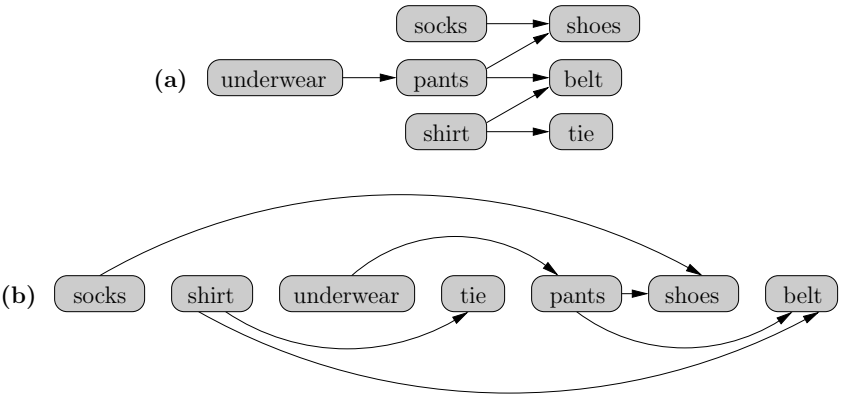

FIGURE 3.6: Illustrations of (a) a directed acyclic graph (DAG) representation of a partially-ordered set, and (b) a topological ordering of the nodes in this DAG.

3.8 Topological Sorting

So far, we have made the fundamental assumption that every pair of input elements is comparable. In this case, we say our input is a *totally-ordered set* or a *total ordering*. In many cases, however, our input may instead be a *partially-ordered set* (*poset*, also called a *partial ordering*), where only certain pairs of input elements can be meaningfully compared. The result of a comparison in a poset can be $<$, $>$, or "incomparable".

The poset in Figure 3.6(a) shows constraints governing the order in which you can put on different articles of clothing. It also shows how we can represent a poset by a *directed acyclic graph*, or DAG. A directed path in this graph between two elements i and j indicates that i is less than j (in our example, this means i must precede j when getting dressed). Our graph cannot have any directed cycles, since if two nodes i and j were on a directed cycle, there would exist directed paths from i to j and also from j to i, leaving it unclear which element is smaller.

In the example above, an edge from underwear to shoes is unnecessary since it is implied by transitivity: underwear is less than pants, which is in turn less than shoes. When we draw the DAG representation of a poset, we typically omit as many implied edges as possible to simplify the diagram. The DAG with the fewest edges that represents a given poset is known as the *transitive reduction* of the poset, and the DAG with all implied edges present is called the *transitive closure*. We will see how to compute both of these in Chapter **??**.

Every DAG can be *topologically sorted* to produce a total ordering of its nodes (known as a *topological ordering*) that is consistent with the partial ordering implied by the DAG. As seen in Figure 3.6(b), all edges point consistently from left to right when we arrange the DAG according to its topological ordering. Many topological orderings may be valid for a given DAG; as an extreme example, any ordering is valid if our DAG has no edges.

Topological orderings have many applications. The instance shown in Figure 3.6 can be viewed as a *scheduling* problem, where nodes represent tasks and edges represent precedence constraints between these tasks; DAGs arise commonly in this setting. A topological ordering here gives a linear task ordering that respects all of the precedence constraints. Just as sorting can be a useful preprocessing step for many algorithms, topological sorting is a common preprocessing step for algorithms dealing with DAGs. We will see several examples of this when we study graphs in greater detail later in the book.

There are several ways to topologically sort a DAG in linear time, which in this case means $O(m + n)$ time if the DAG has n nodes and m edges. We discuss one method here and another, based on a simple graph algorithm called *depth-first search*, in Chapter **??**. It is easy to show that every DAG always has at least one "source" node, with no incoming edge. If not, you could start at any node and walk backward, moving from one node to the next by always stepping backward along an arbitrary incoming edge. Since the graph has a finite number of nodes, this process would eventually revisit a node and close a directed cycle, thereby contradicting the fact that the graph is acyclic. Any source node is safe to place at the beginning of a topological ordering, since no other node needs to come before it. Therefore, we can generate a valid topological ordering by repeatedly finding and removing source nodes. With care, we can implement this in linear time. [Details]

3.9 Sorting Integers

It is finally time to depart from the familiar comparison-based model and consider RAM algorithms for sorting integers in the range $0 \ldots C - 1$. If C is small, this allows us to sort faster than by using comparison-based algorithms.

3.9.1 Counting Sort

Suppose that we have an array $A[1 \ldots n]$ containing only zeros and ones. We can easily sort A in linear time by simply counting the zeros and ones, and then building a new sorted array containing the appropriate number of each.

This idea generalizes easily to the case where A contains integers in the range $0 \ldots C - 1$. Here, we use an auxiliary array $N[0 \ldots C - 1]$, where $N[v]$ counts the number of copies of the value v in A. To build N after initializing its entries to zero, we increment $N[A[i]]$ for every $i = 1 \ldots n$. We can then scan through N to re-build A in sorted order. The resulting sorting algorithm, known as *counting sort*, runs in $\Theta(n + C)$ time, which is linear time as long as $C = O(n)$. With care, we can even implement it in a stable fashion. [Implementation details]

3.9.2 Radix Sort

Using multiple invocations of counting sort, we can build a more powerful sorting algorithm known as *radix sort* that sorts in linear time if $C = O(n^c)$ for some constant c.

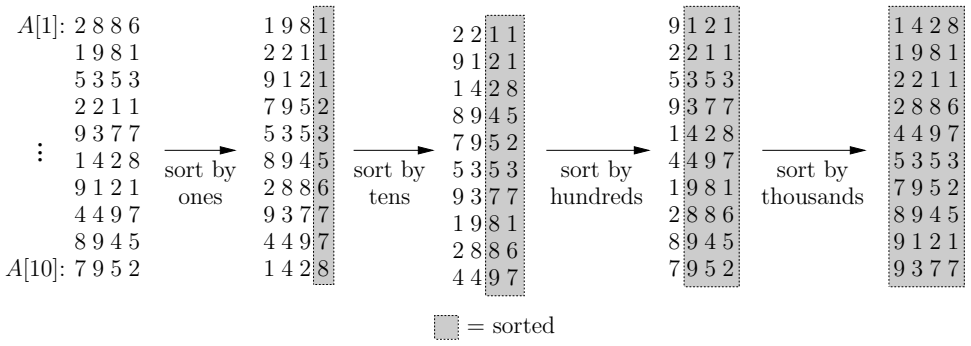

FIGURE 3.7: The operation of radix sort: first perform a stable sort on the least-significant digit, then on the second-least-significant digit, and so on.

Radix sort is illustrated in Figure 3.7. We first write our integers down in some number base, or *radix* (hence the name of the sort), which in our example is base 10. We then sort according to each successive digit position, starting with the least significant. Radix sort actually originated as mechanical technique for sorting stacks of punched cards. Each card contained a number written in binary encoded by a series of punched and non-punched holes, and the cards were repeatedly fed through a mechanical sorting device, once for every digit, and sorted by separating cards with punched digits from those with non-punched digits.

The subroutine we use to perform each of the digit sorts can be any *stable* sorting algorithm, although we use counting sort since it is the natural choice for sorting small integers. If we ideally write our integers in base n, then each digit assumes values from $0 \ldots n-1$ and each individual counting sort runs in $\Theta(n)$ time. Our numbers will have at most $\log_n C$ digits, which is constant as long as $C = O(n^c)$ for some constant c. The total running time is therefore linear as long as $C = O(n^c)$ with c constant[9]. Stability is crucial for our individual digit sorts, since if two numbers agree in their most significant digit (our final sorting pass), we want them to remain ordered properly by their lower-order digits, as they will be at this point thanks to induction.

Problem 48 (Sorting Fractions). Radix sort can sort n integers of magnitude n^c for $c = O(1)$ in linear time. Please show how to sort n fractions $\frac{a_1}{b_1}, \frac{a_2}{b_2}, \ldots, \frac{a_n}{b_n}$ also in linear time, where the a_i's and b_i's are integers in the range $1 \ldots n^c$ for $c = O(1)$. As a hint, can you convert this problem back to one involving sorting small integers? [Solution]

[9]Subtle point: A common assumption with the RAM model of computation is that each word contains $\Theta(\log n)$ bits, which is exactly large enough to hold an integer of magnitude n^c for $c = O(1)$. If the n numbers being radix sorted each fit into a single $\Theta(\log n)$-bit machine word, then radix sort therefore takes only $\Theta(n)$ time. Furthermore, if we plan to sort larger numbers on a machine with word size $\Theta(\log n)$, it takes $\Theta(n \log C / \log n) = \Theta(n \log_n C)$ words to store n numbers of magnitude C, matching the running time for radix sort (so this running time cannot be improved, since it takes the same amount of time just to examine all the input words). The only way to allow for stronger RAM-based sorting algorithms is to consider slightly larger word sizes (which is common practice in the sorting literature). For word sizes slightly larger than $\Theta(\log n)$ bits, it is still unknown if linear-time sorting is possible – see the endnotes for further discussion.

Problem 49 (Sorting Variable-Length Strings). Standard radix sort works from the least significant digit to the most significant. In this problem, we do the opposite, demonstrating a variant called *forward radix sort* to sort a collection of variable-length strings $A_1 \dots A_n$ in standard lexicographical (alphabetical) order. Let us suppose we have a word size of $\Theta(\log n)$ bits, so radix sorting a list of n integer words would take $\Theta(n)$ time. We treat each string A_i as an array of words, from the first "most significant" word down to the "least significant". If we only look at, say, the 7 most significant words of A_i, we may not be able to distinguish A_i from some of the other input strings, if they also agree in these 7 initial words. Let d_i be the minimum number of words (in terms of a prefix of A_i) that we need to examine in order to be able to distinguish A_i from the other input strings. If we let $D = \sum d_i$, then theoretically we should be able to sort $A_1 \dots A_n$ by examining only D total words. Note that this framework also applies to sorting real numbers (say, scaled to the range $[0, 1]$) specified as arrays of digits from most significant down to least significant – here it is crucial to use a forward approach (examining most significant digits first) rather than the backward approach of standard radix sort, since a real number might have an unbounded number of digits.

(a) Note that we typically do not know $d_1 \dots d_n$ at the outset. Show how to compute these values in only $O(nd_{max})$ time, where $d_{max} = \max d_i$. Once we know d_{max}, observe that we can apply standard radix sort to sort $A_1 \dots A_n$ in $O(nd_{max})$ time. [Solution]

(b) The nd_{max} term above is somewhat unsightly. Can you improve the running time for both determining $d_1 \dots d_n$ as well as sorting $A_1 \dots A_n$ to just $O(D)$? [Solution]

Input Sensitivity Versus Insensitivity: A Key Distinction. The running times of counting sort and radix sort are *input-sensitive*, depending on the magnitude C of our input data. Since the running time of counting sort depends directly on C, we say counting sort has a pseudo-polynomial running time, whereas radix sort has a weakly polynomial running time since its running time depends only on $\log C$ (see Section 1.4.6 to review this distinction). All of our other sorting algorithms have strongly polynomial running times (depending only on n) since they run in the comparison-based or real RAM models of computation. There is an important distinction here to be made between these two types of algorithms: input-sensitive RAM algorithms (e.g., counting and radix sort) versus input-insensitive comparison-based or real RAM algorithms (e.g., merge sort and quicksort). For many problems we study, it will be possible to devise both types of algorithms. The choice of which is better depends entirely on the application; neither is inherently "better". With sufficiently small integer input data, input-sensitive approaches may be superior. Otherwise, input-insensitive algorithms can be perhaps aesthetically more appealing and provide some peace of mind, since they do not depend on the magnitude of their input data. The choice between input-insensitive and input-sensitive, or more generally between RAM and real RAM (or comparison-based) algorithms will be a running theme throughout much of this book.

3.10 Advanced Example: Parametric Search

To reinforce the techniques introduced in this chapter, we close with a case study showing how they can be elegantly combined to solve a more challenging problem.

Suppose you take n tests in an algorithms class, earning on the ith test b_i points out of maximum of m_i possible points. As is sometimes common, you are allowed

to drop k tests, with the final grade being determined by a set S containing only the remaining $n-k$ tests:

$$g(S) = \frac{\sum_{i \in S} b_i}{\sum_{i \in S} m_i}.$$

Our goal is to decide which $n-k$ tests to keep so that our grade $g(S)$ is maximized; let g^* denote this maximum value. This is known as the *optimal subset selection problem*, and although it may seem solvable by simply discarding the k tests with the lowest b_i/m_i ratios, this is actually not the case. You may wish to pause and convince yourself of this fact by constructing an appropriate counterexample.

Binary Search on the Answer. As with any new problem, we might approach it by attempting to apply one of our main algorithm design techniques. Divide and conquer, specifically in the form of binary search, yields some useful initial insight. For many optimization problems, it turns out to be much easier to check whether a guess x at the answer is too high ($x > g^*$) or too low ($x < g^*$), than to solve the problem outright. Any time this is the case, we can home in on the optimal solution value g^* quickly using binary search on x. To check if $x < g^*$, we want to know if there exists some subset S of tests with $|S| = n-k$ such that

$$\frac{\sum_{i \in S} b_i}{\sum_{i \in S} m_i} > x,$$

and by rearranging terms, we see that this is the same question as whether or not there exists some subset S with $|S| = n-k$ such that

$$\sum_{i \in S} -m_i x + b_i > 0.$$

If we define $y_i = -m_i x + b_i$, we now simply want to know whether there is a set of $n-k$ tests whose y_i's add up to a positive amount[10]. The answer to this question is yes if and only if the $n-k$ largest y_i's add up to a positive amount, so we could answer it by sorting the y_i's and adding up the largest $n-k$ of them. Better yet, we can use selection to obtain an answer in only $\Theta(n)$ time, by selecting for the kth largest value of y_i and adding up the $n-k$ values above this point. Let us call this $\Theta(n)$-time algorithm for checking whether x is too low or too high $A(x)$.

Thinking Graphically. Our notation $y_i = -m_i x + b_i$ suggests a useful way to visualize the algorithm $A(x)$ and the problem above in general — remember that you can almost always obtain better insight into solving a problem if you can turn it into a picture! If we regard each test i as the line $y_i = -m_i x + b_i$ with slope $-m_i$ and y-intercept b_i, then we can picture our input as a collection of negative-slope

[10]The "binary search on answer" technique is particularly effective for problems like this one involving optimization of a ratio, since through algebraic term re-arrangement it often reduces the problem to one having a simpler (non-ratio) objective. For example, given a numeric array $A[1 \ldots n]$, suppose we want to find a window $i \ldots j$ of length at least k with the largest average value (without the length $\geq k$ constraint, the problem would be trivial, the answer being just the single largest element in A). To test if a guess x for the answer is too small, we ask whether there exists a window $i \ldots j$ such that $\frac{1}{j-i+1}(A[i]+\ldots+A[j]) > x$, otherwise written as $A[i]+\ldots+A[j] > x+\ldots+x$ (with $j-i+1$ terms), which we further simplify to $(A[i]-x)+\ldots+(A[j]-x) > 0$. Defining $B[i] = A[i]-x$, we now want to know whether there exists a subarray of B of length at least k whose sum is positive, which we can answer by finding the maximum sum of a subarray in B of length at least k — a problem that is easy to solve in $\Theta(n)$ time (see, e.g., Section 8.1.1). We have therefore effectively replaced "average" with "sum" in our objective.

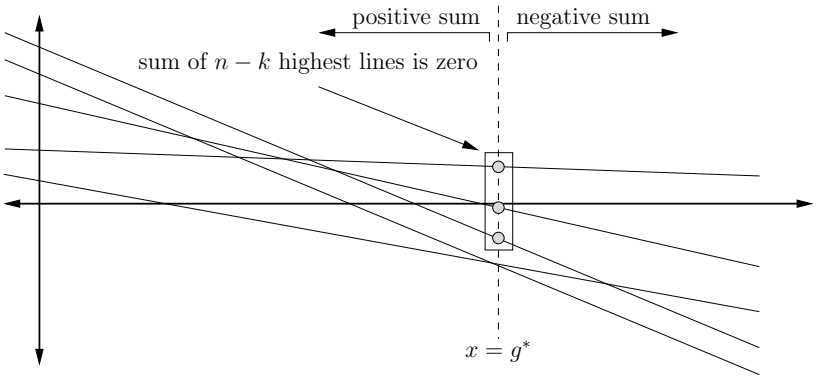

FIGURE 3.8: Visualizing the optimal subset selection problem as the geometric problem of finding the rightmost value of x at which the y values of the $n-k$ highest lines is still nonnegative.

lines as in Figure 3.8. In this geometric context, we want to find the value of x at which the sum of the y values of the $n-k$ highest lines at evaluated at x is zero. Since our lines have negative slope, observe that the sum of the y values of the $n-k$ highest lines decreases as we move x to the right, and increases as we move x to the left. This monotonicity is what makes binary search possible.

Solving the Problem by Already Knowing the Answer. Binary search on x to find g^* will work fine in practice, although this approach still leaves something to be desired, as it only converges to the optimal solution over time, rather than giving an exact answer after a number of steps polynomial in n. On the real RAM, it could theoretically even take infinitely long to converge if g^* is irrational. Let us therefore consider a slightly different approach. Suppose we magically knew the $n-k$ highest lines when evaluated at $x = g^*$. Observe that this is enough to compute g^* in $\Theta(n)$ time, since we can add together the linear functions representing these $n-k$ lines to get a single linear function of x, set it equal to zero, and solve for x.

Now all we need to do is compute the highest $n-k$ lines at $x = g^*$. Normally, we would do this in $\Theta(n)$ time by using quickselect to select for the kth highest line at $x = g^*$, and then partitioning on this line to obtain the $n-k$ lines above it. This may all seem futile, however, since we don't know g^*, so we don't know what value of x to plug in when evaluating our lines. Accordingly, let us keep the y_i's in the form of generic linear functions (e.g., $-5x+3$), while running the algorithm above. This doesn't cause problems until we reach a comparison, where we are now comparing two linear functions, such as $-5x+3$ versus $-7x+9$. The result of this comparison depends on x; here, the first function is smaller if $x < 3$ and larger if $x > 3$. Remembering that we are planning to evaluate these functions at $x = g^*$, all we need to know to resolve the comparison is therefore whether or not $g^* < 3$, and we can answer this question by running $A(3)$.

Our approach now makes sense. As we run our high-level selection algorithm with the y_i's represented generically as linear functions, we pause at every comparison run an invocation of $A(x)$ to test the "breakpoint" value of x that is necessary to

resolve the comparison. Since every comparison of the high-level $\Theta(n)$ algorithm invokes a lower-level $\Theta(n)$-time algorithm, the total running time is $\Theta(n^2)$.

A Clever Use of Divide and Conquer. One final idea gives a dramatic improvement in running time. Our high-level algorithm based on quickselect performs a number of partition operations, each one comparing all the elements in an array against a specific pivot element. Since these comparisons don't depend on eachother (they could in theory be done in parallel), we can therefore use divide and conquer based on selection and binary search to resolve n such comparisons in only $\Theta(n)$ time plus only $O(\log n)$ invocations of A (instead of n invocations as before). This drops our total running time to only $O(n \log n)$ in expectation. [Full details]

The technique above, known as *parametric search*, is useful in a wide range of problems, particularly in computational geometry. It is also an interesting example of how ideas from parallel computation can be used in novel ways to speed up sequential (non-parallel) algorithms. For our purposes, it serves as a wonderful example of what one can achieve using smart combinations of basic design techniques. See the endnotes for further references regarding the optimal subset selection problem, as this problem even admits a $\Theta(n)$ solution!

3.11 Closing Remarks and Additional Problems

The ideal sorting algorithm would be comparison-based, stable, in-place, deterministic, run in $O(n \log n)$ worst-case time (faster for nearly-sorted inputs), and require only $O(n)$ memory writes. So far we only how to achieve certain specific combinations of these features, so despite the fact that sorting is such a fundamental problem, it continues to challenge even the best computing researchers.

Sorting is a key step in many algorithms (e.g., "sweep line" algorithms in computational geometry), and we will see problems involving sorting throughout this book. When we study data structures in the next few chapters, we will learn even more ways to sort, using data structures like priority queues and binary search trees.

Problem 50 (Sorting as a Means of Grouping). One common use of sorting during preprocessing is simply to group similar elements together. As a nice example of this approach in action, please describe an $O(n^2 \log n)$ algorithm that takes n points in the 2D plane as input and counts the number of parallelograms whose corners all come from the point set. Can you count rectangles as well (not necessarily aligned with the x and y axes)? [Solution]

Problem 51 (Divide and Conquer Practice). This problem contains a collection of nice practice problems that can be solved elegantly using the divide and conquer principle.

(a) **Repeated Squaring.** Suppose we want to compute A^k, where A is an $n \times n$ matrix. We could do this by starting with A and successively multiplying by A for $k-1$ iterations. However, since matrix multiplication is a time-consuming operation (the straightforward approach for multiplying two $n \times n$ matrices gives has an $O(n^3)$ running time), we would like to use as few multiplications as possible. Please show how to compute A^k using only $O(\log k)$ matrix multiplications and additions (note that matrix additions are much less costly — these only take $O(n^2)$ time). As a hint, remember

that the technique you are developing goes by the name of *repeated squaring*. Despite its simplicity, this method is quite powerful, and it is commonly used in practice for raising any complicated object (e.g., a matrix or a polynomial, or any other object for which multiplication is costly) to a large power. [Solution]

(b) **Median Among Two Sorted Arrays.** You are given two sorted arrays $A[1 \ldots n/2]$ and $B[1 \ldots n/2]$ as input. Please show how to compute the median among all the n elements contained in both arrays in $O(\log n)$ time. For a challenge, what is the best performance guarantee you can offer (in terms of n and k) if A and B contain k and $n-k$ elements rather than $n/2$ elements each? What if you want to find the median element among k sorted arrays each containing n/k elements? [Solution]

(c) **Find the Missing Element.** You are given two arrays: A, containing n elements, and B, containing $n-1$ of the elements present in A. Design a comparison-based algorithm running in $\Theta(n)$ time that determines which element in A does not appear in B. How quickly can you solve the more general problem variant where k elements are left out of B? What about the similar problem where B contains $n+1$ elements — including all of the elements in A, but with one of them occurring twice — where we wish to locate the duplicated element? [Solution]

(d) **Cyclic Shift of an Increasing Sequence.** Suppose $A[1 \ldots n]$ is obtained by performing a k-step right cyclic shift of a length-n array whose elements form a strictly increasing sequence (a k-step right cyclic shift moves every element to an index k steps ahead, with the k final elements "wrapping around" and becoming the k first elements). Given A, how can you determine k in $O(\log n)$ time? [Solution]

(e) **Monotone Matrix Searching.** Let A be an $n \times n$ matrix whose entries are distinct numbers, and let $M(i)$ denote index of the minimum element in row i. We say A is *monotone* if $M(1) \leq M(2) \leq \ldots \leq M(n)$. Please give an $O(n \log n)$ algorithm for computing $M(1) \ldots M(n)$ in an $n \times n$ monotone matrix[11]. [Solution]

(f) **Approximate Binary Search.** Suppose you want to find an unknown value v in the range $1 \ldots n$. If v is an integer, we can of course binary search for v in $O(\log n)$ time by first guessing $n/2$, then seeing if this guess is too high or too low, and successively halving our search range in each step. Suppose, however, that v is not necessarily an integer, and that we are content to find an approximate solution within the range $(1-\varepsilon)v \ldots (1+\varepsilon)v$, where ε is some constant. As it turns out, we can solve this problem in only $O(\log \log n)$ time by using the geometric mean rather than the arithmetic mean as our guess for each step in the binary search. That is, when we are searching the interval $[a, b]$, our next guess will be $\sqrt{ab}$. Please prove that this does indeed give us an $O(\log \log n)$ running time. As a hint, consider problem 8. [Solution]

(g) **Longest Line of Sight.** You are given an array $H[1 \ldots n]$ containing distinct numbers that describe the heights of n people standing in a line. We say person i and person $j > i$ can see each-other if $H[i]$ and $H[j]$ are both larger than all of the elements in $H[i+1 \ldots j-1]$. Please show how to locate a farthest pair of individuals i and j that can still see each-other in $O(n \log n)$ time[12]. [Solution]

(h) **Printing a Linked List Backwards.** You are given as input a pointer to the first element of an n-element linked list (without being told n), and asked to print out the contents of the list in reverse order without physically modifying the elements of the list in any way. This is trivial to do in $\Theta(n)$ time if we allow $\Theta(n)$ extra space, but somewhat more interesting if we wish to use much less auxiliary memory while still maintaining a fast running time. Please describe an $O(n \log n)$ time solution using

[11] In Section 11.5, we will see monotone matrices with even more structure, known as *totally monotone* matrices, for which we can actually compute $M(1) \ldots M(n)$ in only $\Theta(n)$ time (for the case of a monotone matrix there is actually an $\Omega(n \log n)$ worst-case lower bound in the comparison model). As we will see, this result has surprisingly many applications!

[12] In problem 131, we improve the running time to $\Theta(n)$.

only $O(\log n)$ auxiliary words of memory. Can you generalize your solution to run in $O(kn)$ time using $O(kn^{1/k})$ space for any $k = 1 \ldots \log n$? Observe that $k = 1$ gives the trivial solution where we use $\Theta(n)$ extra space, and that $k = \log n$ leads to the time and space bounds of $O(n \log n)$ and $O(\log n)$ above. [Solution]

(i) **Finding a Local Minimum.** Let $f(x)$ be a function defined over all integers x in the range $1 \ldots n$. We say x is a local minimum of f if $f(x) \leq f(x-1)$ and $f(x) \leq f(x+1)$, where we treat $f(0) = f(n+1) = +\infty$. Please show how to compute a local minimum of f by evaluating f at only $O(\log n)$ points. Next, let $g(x, y)$ be a function defined over integers x and y in the range $1 \ldots n$. We say (x, y) is a local minimum if $g(x, y)$ is no larger than any of the four neighboring values $g(x \pm 1, y)$ and $g(x, y \pm 1)$ (again, we treat $g(x, y)$ as being infinite if x or y is not in the range $1 \ldots n$). Please show how to compute a local minimum of g with only $O(n)$ function evaluations. [Solution]

(j) **Cake Cutting.** Suppose you want to divide up a cake among n people so that every person feels like he or she has received a fair-sized piece. This is easy if everyone values every part of the cake equally, since the solution in this case is just to give $1/n$ of the cake to everyone. However, in practice we might find that different individuals might have non-uniform preferences over different parts of the cake (e.g., I might like the side of the cake with more frosting, and you might prefer the side of the cake with more sprinkles on top). To be somewhat more formal, let us consider the unit interval $[0, 1]$ as a long one-dimensional "cake" Each of our $i = 1 \ldots n$ participants must be served a piece of cake that corresponds to a contiguous piece of this interval. The preferences of participant i are described by a valuation function $f_i(x)$ that specifies his/her value for the interval $[0, x]$. Each $f_i(x)$ function is continuous, monotonically increasing, and ranges from $f_i(0) = 0$ up to $f_i(1) = 1$ (i.e., the whole cake is worth 1 unit to each participant). Note that we can find the value of any subinterval $[a, b]$ by taking $f_i(b) - f_i(a)$. In addition to $f_i(x)$, suppose we also have access to its inverse function $f_i^{-1}(y)$, so for example $f_i^{-1}(1/2)$ would tell us the unique point x such that exactly half of the value for participant i lies in $[0, x]$ and the other half lies in $[x, 1]$. Suppose that it takes $O(1)$ time to evaluate $f_i(x)$ or $f_i^{-1}(y)$. Design an $O(n \log n)$ time algorithm that computes a subinterval $[a_i, b_i]$ for each participant i such that $f_i(b) - f_i(a) \geq 1/n$ (i.e., each participant receives what he or she perceives to be a "fair" piece of the cake). [Solution]

(k) **Counting Distant Pairs of Points.** You are given n points $(x_1, y_1) \ldots (x_n, y_n)$, where the x_i's and y_i's are integers in the range $1 \ldots n$. Two points (x_i, y_i) and (x_j, y_j) are said to be (a, b)-distant if $|x_i - x_j| \geq a$ and $|y_i - y_j| \geq b$. Given a and b as input, please describe an $O(n \log n)$ divide-and-conquer algorithm for counting the number of pairs of (a, b)-distant points. For a warm-up, you may want to consider $a = b = n/2$. [Solution]

(l) **The "Firing Squad" Problem.** This is a classical problem in parallel algorithm design that has an elegant solution based on the principle of divide and conquer. Suppose we have n processors connected in a line. Each processor is synchronized with a global clock, and each time the clock ticks the processor can perform $O(1)$ work as well as send $O(1)$ information to its neighbors (so it takes $n - 1$ time steps for a message from one end of the line to propagate down to the other end). However, each processor is somewhat limited in that it only has a constant number of bits of memory, independent of n. This prohibits processors from doing things like counting to n (which would require $\log_2 n$ bits of memory). Our goal is to design an algorithm to run continuously on these processors (each processor should run the same algorithm) such that if we input a special message to one of the endpoint processors at some time, all processors should simultaneously enter a special state (e.g., they should all "fire") at some point $O(n)$ time steps later. Can you devise a simple algorithm that accomplishes this goal? [Solution]

Problem 52 (Frequently-Occurring Elements). We can easily solve the problem of finding the most-frequently-occurring element in an n-element array in $O(n \log n)$ time by sorting, and we also have an $\Omega(n \log n)$ worst-case lower bound on this problem in the comparison model via a simple reduction from the element uniqueness problem. However, if the most-frequently-occurring element appears more than $n/2$ times in the array, we can solve this special case in only $\Theta(n)$ time (how do we do this?) This motivates us to try and design an algorithm for finding the most-frequently-occurring element whose running time scales gracefully between $\Theta(n)$ and $\Theta(n \log n)$, depending on the number of occurrences k of the most-frequently-occurring element. Please show how to solve this problem in $O(n \log(n/k))$ time, and if you are feeling ambitious, try to prove a matching lower bound in the comparison-based model. [Solution]

Problem 53 (k-Way Merging). Consider the problem of merging k previously-sorted lists containing n total elements. There are several ways to solve this problem in $O(n \log k)$ time. For example, one can perform the standard iterative merging algorithm using a data structure such as a binary heap or balanced binary search tree to maintain the k leading elements in our lists; in every iteration, we select the minimum such element in $O(\log k)$ time, adding it next to the merged sequence. Please comment on how we might instead solve this problem using a divide-and-conquer approach. Next, argue that there is a matching worst-case lower bound of $\Omega(n \log k)$ in the comparison-based model for any algorithm that performs k-way merging of n elements. [Solution]

Problem 54 (Sorting A Sequence Drawn from a Small Universe). Suppose we have array of n elements drawn from a universe of only $k \leq n$ distinct elements, and we wish to sort this array using a comparison-based sorting algorithm. If k is small we would hope to improve on the $O(n \log n)$ running time for generic sorting algorithms.

(a) Can you modify merge sort to solve this sorting problem in only $O(n \log k)$ time? Your algorithm will not be told the value of k. [Solution]

(b) Show that randomized quicksort applied to this problem runs in $O(n \log k)$ time with high probability, even though it also does not know the value of k. [Solution]

(c) Argue that there is a lower bound of $\Omega(n \log k)$ on the worst-case running time of any comparison-based algorithm that solves this problem, even if the algorithm knows the k distinct values appearing throughout the array. [Solution]

(d) Suppose you are given two strings (arrays) $A[1 \ldots n]$ and $B[1 \ldots n]$ whose characters (elements) are drawn from an *alphabet* Σ (e.g., the letters A through Z). We wish to detect whether or not A and B are *anagrams* — that is, whether or not you can transform A into B by permuting its elements. If $k = |\Sigma|$, show that in the comparison-based model of computation one can detect anagrams in $O(n \log k)$ time, and also show that there is a matching lower bound of $\Omega(n \log k)$ on the worst-case running time of any anagram detection algorithm. [Solution]

Note that another nice solution for sorting in $O(n \log k)$ time will become apparent once we learn in Chapter 6 to use balanced binary search trees as "maps" — in that case, we insert our n values in to a balanced binary search tree where each element is augmented with a frequency count (this takes $O(n \log k)$ time, since the tree contains k elements), and then we traverse the tree to enumerate its contents in sorted order).

Problem 55 (The Zero/One Sorting Theorem). The well-known *zero/one sorting theorem* is very useful for proving correctness of complicated sorting algorithms. It states that a comparison-based sorting algorithm is correct if and only if it correctly sorts sequences consisting of just zeros and ones. Therefore, in order to argue correctness of a complicated sorting algorithm, it suffices to argue that it sorts any sequence of zeros and ones. Please give a short proof of the zero/one sorting theorem. [Solution]

Problem 56 (Chain and Block Sorting). Let us think of an array $A[1 \ldots n]$ as k interleaved "chains" of elements. The first chain is $A[1], A[k+1], A[2k+1], \ldots$, the second is $A[2], A[k+2], A[2k+2], \ldots$, and so on. We say an array $A[1 \ldots n]$ is *k-chain-sorted* if each of its k chains is sorted. We can also think of an array $A[1 \ldots n]$ in terms of n/k "blocks" of size k. The first block is $A[1 \ldots k]$, the second is $A[k+1 \ldots 2k]$, and so on. We say A is *k-block-sorted* if each of its n/k blocks is sorted.

(a) What is the fastest possible running time (in the comparison model) for making an array k-chain-sorted or k-block-sorted? What is the fastest possible running time (also in the comparison model) for sorting an array that is already k-chain-sorted and k-block-sorted? [Solution]

(b) Take an n-element array that is k-chain-sorted and k-block-sorted. For notational simplicity, let $A = \min(k, n/k)$ and let $B = \max(k, n/k)$. Suppose we want to search for a particular element based on its value. Give algorithms that perform this task in $O(A \log B)$ time (fairly easy), $O(A+B)$ time (slightly more difficult), and $O(A \log(1+B/A))$ time (a bit more difficult). Note that the last running time dominates both of the other two. [Solution]

(c) An n-element array that is both k-chain-sorted and k-block-sorted for $k = \sqrt{n}$ can be visualized as the row-order contents of a $\sqrt{n} \times \sqrt{n}$ matrix of elements whose rows and columns are all sorted. In the preceding problem, we showed how to search in such a structure in $O(\sqrt{n})$ time. How quickly can you search in the d-dimensional generalization of this structure? For example, if $d = 3$, then your data would be stored in a 3-dimensional array of size $n^{1/3} \times n^{1/3} \times n^{1/3}$ that is sorted along each dimension. [Solution]

(d) Show that if we k-block-sort (by sorting each of its blocks independently) an array that is already k-chain-sorted, the array remains k-chain-sorted, and vice-versa. Show also that for any k and k', if we k'-chain sort (by sorting each of its chains independently) an array that is already k-chain-sorted, then the array stays k-chain-sorted. Is the same true for k-block-sorting and k'-block-sorting? [Solution]

Problem 57 (Shell Sort). Shell sort, named after its creator Donald Shell, is a method for speeding up insertion sort by allowing elements to be moved longer distances initially. In the context of the preceding problem, it involves making an array k-chain-sorted (by calling insertion sort on each of its k chains independently) for decreasing values of k coming from a so-called *increment sequence*. For example, suppose we use the increment sequence $\{1, 3, 7, 15, \ldots\}$ of integers one less than powers of two, so we start with k being the largest increment in this sequence less than n, and end with $k = 1$. Increment sequences always end with $k = 1$, since this ensures the final array is sorted (note that 1-chain-sorted means the same thing as sorted). Shell sort runs in $O(n^{1.5})$ time for the increment sequence above, but a better sequence (at least in theory, although sometimes not in practice) is the set containing all integers of the form $2^x 3^y$, of which there are $\Theta(\log^2 n)$ elements at most n. Using the result from the last part of the preceding problem, we know that when it comes time to $2^x 3^y$-chain-sort our array, it will already be $2^{x+1}3^y$-chain-sorted and $2^x 3^{y+1}$-chain-sorted. Show that this implies that k-chain-sorting for each increment k takes only $\Theta(n)$ time, leading to an overall running time of $\Theta(n \log^2 n)$, which is nearly the best attainable for Shell sort (see the endnotes for further details). As a hint, remember that insertion sort runs quickly when there are few inversions in the arrays being sorted. [Solution]

Problem 58 (Selection in a Sorted Matrix). Building on your solution to problem 56 above, please now consider the problem of selecting the kth largest element from an $n \times n$ matrix whose rows and columns are both sorted. For a challenge, see if you can solve this problem in only $\Theta(n)$ time. Please assume for simplicity that all entries in the matrix are distinct. As a hint, try partitioning the matrix into small blocks (e.g.,

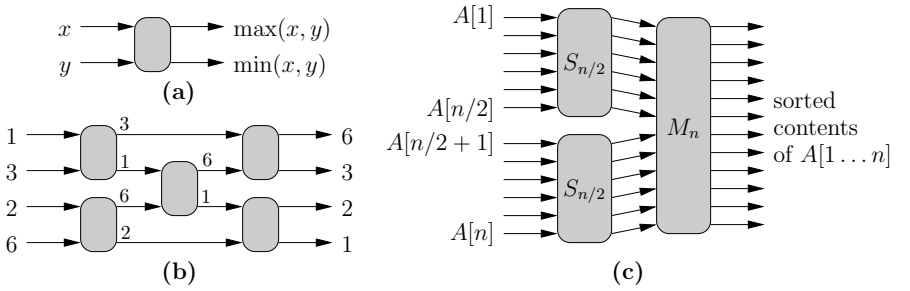

FIGURE 3.9: Diagrams of (a) a comparator, (b) a 3-stage sorting network that sorts 4 inputs, and (c) an n-element sorting network that implements merge sort: The blocks labeled $S_{n/2}$ recursively sort inputs of length $n/2$, and the block labeled M_n merges their output into a sorted list of length n.

2×2 or 3×3 in size), and construct a smaller matrix by replacing each block with its minimum or maximum; by recursively solving a selection problem in this smaller matrix, you should be able to exclude from consideration many of the elements in the original matrix. [Solution]

Problem 59 (Parallel Sorting Networks). In this problem we briefly delve into the realm of parallel sorting algorithms. A common model for parallel sorting is the *sorting network*, a circuit taking n numbers as input that produces their sorted ordering as output. An example of a sorting network is shown in Figure 3.9(b). Sorting networks are built from blocks called *comparators* (Figure 3.9(a)) that compare two input numbers and output the numbers in sorted order. A sorting network is built from a series of *stages* (or *layers*), each of which contains several comparators. However, each stage can be at most one comparator "deep" (i.e., a signal may not pass through two different comparators in the same stage). The goal of a sorting network is to minimize the number of stages, as this corresponds to the running time of the network. Although it does not seem possible to map any of our most common $O(n \log n)$ comparison-based sorting algorithms (e.g., merge sort and quicksort) to an efficient parallel sorting network, there are several elegant approaches for constructing efficient sorting networks. We discuss two of these below.

(a) **Batcher's Odd-Even Merge Sort**. Suppose we wish to merge two length-$n/2$ sorted arrays A and B into a single length-n sorted array C. Consider the following somewhat strange "odd-even" recursive approach for performing this merge: the odd elements of C (i.e., $C[1]$, $C[3]$, etc.) are obtained by recursively merging the odd elements of A and B, and the even elements of C are obtained by recursively merging the even elements of A and the even elements of B. After running these recursive merges and filling up C, we make one final postprocessing pass in which we compare successive pairs of elements in C: $C[1]$ versus $C[2]$, $C[3]$ versus $C[4]$, and so on, swapping any pairs that are out of order. Please use the zero/one sorting theorem to prove that this algorithm merges correctly. Next, show how to build a sorting network based on this merge operation that requires $O(\log^2 n)$ total stages. [Solution]

(b) **Bitonic Sorting**. Another way to merge two sorted length-$n/2$ sequences is to concatenate the first with the reversal of the second and then to sort the resulting *bitonic* sequence. We call a sequence bitonic if it increases up until some point and then decreases, or if it is a "cyclic shift" of such a sequence. Although it may seem like we're moving in the wrong direction by converting a merging problem back into a sorting problem, it turns out that bitonic sorting is a natural and easy problem on a sorting

network. Suppose $A[1 \ldots n]$ contains a bitonic sequence. In a single stage of our sorting network, we place comparators between $A[1]$ and $A[n/2+1]$, $A[2]$ and $A[n/2+2]$, and so on. After executing this stage, show (using the zero/one sorting theorem) that the elements in $A[1 \ldots n/2]$ must all be no larger than the elements in $A[n/2+1 \ldots n]$, and that moreover, these two length $n/2$-subarrays must also be bitonic. As a consequence, we can now continue to sort these two half-sized subproblems recursively in parallel. Show that merging according to this strategy requires $O(\log n)$ stages, and that this leads to a parallel version of merge sort that requires $O(\log^2 n)$ total stages. [Solution]

Problem 60 (The Post Office Problem). We know the locations of n businesses in the downtown section of our city — these locations are simply (x, y) coordinates in the plane. Since the streets in downtown are shaped like a grid, we measure distances using the L_1, or "Manhattan" metric, in which the distance between two points (x_1, y_1) and (x_2, y_2) is given by $|x_1 - x_2| + |y_1 - y_2|$.

(a) Suppose we wish to choose the coordinates of a post office such so as to minimize the sum of distances between the post office and each business. Show how to find a suitable location for the post office in $\Theta(n)$ time. If you want to start with a simpler problem, consider first the one-dimensional variant where the businesses are points on a number line. Can you generalize your solution to work in higher dimensions? [Solution]

(b) Let us now assign weights $w_1 \ldots w_n$ to the businesses, indicating the relative important of the post office being close to each business. We now wish to find a location for the post office which minimizes the weighted sum of distances to the businesses. Show how to solve this problem in $\Theta(n)$ time. [Solution]

Problem 61 (Counting Inversions). In this problem we will attempt to count the inversions in an n-element array[13].

(a) Modify the merge sort algorithm to count the total number of inversions in an array while still running in $O(n \log n)$ time. Try to extend this algorithm to count the number of inversions *per array element*. We define the number of inversions for an element $A[j]$ as the number of elements preceding it in the array that have larger values than $A[j]$. We only consider preceding elements so we count each inversion exactly once — summing up all of the per-element inversion counts should therefore give the total number of inversions. [Solution]

(b) Suppose we would like to efficiently count inversions in an array of integers in the range $0 \ldots C-1$. Show first that if $C = O(1)$, then we can do this in linear time. If $C = O(n^c)$ for $c = O(1)$, then we can sort in linear time via radix sort but it seems difficult to exactly count inversions in linear time. However, we can approximate the number of inversions in linear time. Consider summing up for each element in our array the distance between its current index in the array and its rank (i.e., its index within the array once the array is sorted). The resulting measure, F, is known as *Spearman's Footrule*. Show that we can compute this quantity in linear time, and that $I \leq F \leq 2I$ where I denotes the number of inversions in our array. [Solution]

Problem 62 (Counting Subarrays). You are given as input an array $A[1 \ldots n]$ of real numbers and a target value v.

(a) Please describe how to count the number of subarrays $A[i \ldots j]$ with *sum* at least v in $O(n \log n)$ time. [Solution]

[13] We will see additional solutions to this problem in problems 96 and 122.

(b) Please describe how to count the number of subarrays $A[i \ldots j]$ with *median* at least v in $O(n \log n)$ time, or better yet, $O(n)$ time (for a subarray of even size, this means that at least half its elements must be no smaller than v). [Solution]

(c) Please describe how to count the number of subarrays $A[i \ldots j]$ with *mean* at least v in $O(n \log n)$ time. [Solution]

Problem 63 (Rank Aggregation). There are many situations in practice where we might want to solve a *rank aggregation* problem that involves finding an ordering of n elements that is "reasonably close" to a set of k different orderings. For example, in an athletic competition with n athletes, we may have k judges who each independently rank the athletes, after which we need to generate a single ranking that agrees the greatest extent with the rankings of all k judges. Alternatively, when conducting a web search, it may be more robust to somehow combine the rankings of n pages by k different search engines, rather than to use the rankings returned by only a single search engine. Suppose, given k ranked lists $L_1 \ldots L_k$ on the same set of n elements, that we wish to construct a single ranked list L^* minimizing $\sum_i I(L_i, L^*)$, where $I(a, b)$ denotes the number of inversions between lists a and b. This problem is NP-hard, although it can be approximated well (see also the endnotes for more details). Here, please show that if we choose one of the k input orderings uniformly at random, then this gives a 2-approximate solution. This fact implies that if we take the best of the k orderings, we also obtain a 2-approximate solution, since the minimum of a set of numbers is always no larger than the average. You can either solve this problem directly or, if you prefer, using the result of problem 35. In the second case, all you need to show is that the inversion distance between two orderings is a *metric*: for any three orderings a, b, and c, the triangle inequality $I(a, b) + I(b, c) \geq I(a, c)$ holds. We discuss an alternate 2-approximate solution based on network flow algorithms in problem **??**. [Solution]

Problem 64 (Fun With Intervals). You are given n intervals $[a_1, b_1] \ldots [a_n, b_n]$ on the number line. For simplicity, assume all interval endpoints are distinct (this assumption is not fundamentally required for any of the results below).

(a) Two intervals can be related in 3 possible ways: they can *nest* (one interval within the other), they can be *disjoint* (not overlapping at all), or they can *cross* (overlapping but not nesting). A set of intervals is said to be *laminar* if there are no crossing pairs of intervals; that is, if two intervals overlap at all, they must nest. Laminar intervals exhibit a sort of "balanced parenthesis" structure. Give an $O(n \log n)$ algorithm for testing whether or not a set of n intervals is laminar. [Solution]

(b) For a challenge, see if you can prove an $\Omega(n \log n)$ worst-case lower bound on the running time of any deterministic comparison-based algorithm that solves part (a). [Solution]

(c) Suppose for each interval in our set we wish to compute its *containment count* (the number of intervals contained within the specified interval), its *inclusion count* (the number of other intervals containing the specified interval) as well as its *overlap count* (the number of other intervals overlapping the specified interval, not counting those containing or contained within the interval). Show how to compute these quantities for each of the n input intervals in $O(n \log n)$ time. [Solution]

Problem 65 (Nesting Sets). Suppose we have a collection of n sets $S_1 \ldots S_n$, each containing some subset of the elements $\{1, 2, \ldots, m\}$. For each set, we are given as input a list of the elements it contains. Two sets S_i and S_j are said to be *disjoint* if $S_i \cap S_j = \emptyset$, to *nest* if $S_i \subseteq S_j$ or $S_j \subseteq S_i$, and to *cross* otherwise. If every pair of sets is disjoint or nesting, we say our collection of sets is *laminar*. Please design an $O(mn)$ algorithm for checking whether a collection of sets is laminar. Can you improve the running time to $O(m + n + k)$, where k is the total number of elements in all the input sets? [Solution]

Problem 66 (Average and Median Distance on a Line). In this problem we investigate the computation of statistical information about the set of $\binom{n}{2}$ distances between n points on a number line. As input, you are given the locations $x_1 \ldots x_n$ of these points (not necessarily sorted).

(a) Give an $O(n \log n)$ algorithm for computing the average distance among all pairs of points. [Solution]

(b) Give a randomized algorithm running in $O(n \log n)$ expected time that computes the kth largest of all $\binom{n}{2}$ pairwise distances. [Solution]

(c) For a challenge, can you find a deterministic $O(n \log n)$ algorithm for the problem from part (b)? [Solution]

Problem 67 (Sorting with Substring Reversals). The *reversal distance* between sequences A and B is the minimum number of substring reversals required to transform A into B (often we consider B to be the sorted order of A, asking us to sort A with a minimum number of substring reversals). The problem of computing reversal distance (or equivalently, sorting by reversals) has application in computational biology, where the reversal distance between two strands of DNA gives some indication of their evolutionary similarity. It is NP-hard[14], even in the special case where we are only allowed to reverse prefixes of our sequence, affectionately known as the *pancake flipping* problem since it models the situation where we are trying to sort a stack of pancakes of different sizes using a spatula with which we can only flip over a group of pancakes at the top of the stack[15].

(a) A simple approach for pancake flipping is to place the largest pancake at the bottom of the stack (if it isn't already there) by first flipping it up to the top, and the reversing the entire stack. We then proceed to place the second-largest pancake, and so on, only moving pancakes that need to be moved. Please argue that this is actually a 2-approximation algorithm. As a hint, a useful concept here is the notion of a *breakpoint*, which exists between two pancakes if they are not neighbors in the final sorted sequence. [Solution]

(b) Now consider the general problem of sorting with substring reversals. A natural greedy algorithm for this problem is to reverse, repeatedly, a substring that reduces the number of breakpoints in our sequence as much as possible. Since a reversal can only affect breakpoints at its endpoints, the best we can hope to do is reduce the number of breakpoints by 2. The worst we can do is fail to reduce the number of breakpoints at all (so we need to be careful in proving that the algorithm actually terminates). As a tie-breaking mechanism, if the algorithm can only manage to decrease the number of breakpoints by 1, then it should make its choice in such a way that it leaves a decreasing *strip* if possible. A strip is just a substring of elements between any two consecutive breakpoints, which can be either increasing or decreasing. For a challenge, prove that the greedy algorithm is a 2-approximation algorithm. As a starting point, show that the greedy algorithm uses only $B - 1$ reversals to sort a sequence with B breakpoints, as long as it has at least one decreasing strip (hence, if there are not any decreasing strips initially, our first reversal will create one, so we will use at most B reversals in total). [Solution]

[14] The general problem of computing the reversal distance between two strings is known to be NP-hard, although somewhat surprisingly its "signed" variant can be solved in polynomial time by a rather complicated algorithm (further references can be found in the endnotes). The signed variant is particularly relevant to some biological situations where elements of our sequence each have an inherent notion of directionality; for example, by reversing the substring BCD in $ABCDE$ we would end up with $AD'C'B'E$ where X' denotes the element X oriented in the reverse direction. Our sequence of reversals in the signed case must leave every element oriented in the forward direction. In this problem, we consider only the "unsigned" variant.

[15] Its signed variant is known as the "burned pancake" problem, where each pancake has a burned side, and all of these need to end up facing downward.

Problem 68 (Perfect Shuffles and In-Place Matrix Rearrangement). When playing cards, a *perfect shuffle* involves taking the top and bottom halves of a deck of cards and interleaving the two. If we think of the deck of cards as an array of length n having elements $A[1 \ldots n/2]$ followed by $B[1 \ldots n/2]$, then after a perfect shuffle the array would contain $A[1], B[1], A[2], B[2], \ldots, A[n/2], B[n/2]$. Here, we design elegant divide-and-conquer algorithms for perfect shuffles and related problems that operate in place.

(a) Design in-place $O(n \log n)$ algorithms for performing (i) a perfect shuffle and (ii) the inverse of a perfect shuffle. As a hint, consider problem 45. [Solution]

(b) Suppose we store an $m \times n$ matrix in *row-major* form — as an array of length $N = mn$ in which we list the elements of the matrix one row at a time from left to right. By taking the *transpose* of such a matrix, we convert it into *column-major* form — an array of length N listing the columns of the matrix one at a time, each one from top to bottom. The perfect shuffle is equivalent to computing the transpose of a $2 \times n/2$ matrix, and more generally the transpose of an $m \times n$ matrix can be seen as an m-ary perfect shuffle, where we divide a sequence into m equal blocks and then interleave their contents. Using the result from (a) as black box, show how to compute the transpose of an arbitrary matrix in place in $O(N \log^2 N)$ time. [Solution]

(c) Consider again an $m \times n$ matrix stored in row-major form in an array of length $N = mn$. The in-place *matrix reblocking* problem involves dividing up the matrix into $a \times b$ blocks. That is, we view the matrix as an $m/a \times n/b$ matrix of blocks, each block having size $a \times b$. In order to reflect this restructuring of the matrix, we wish to rearrange its elements in memory so that the blocks are listed in row-major form, and the contents of each block are also listed in row-major form. This allows us to more easily apply divide and conquer algorithms that operate by partitioning a matrix into blocks. Please show how to reblock a matrix, in place, in $O(N \log^2 N)$ time. [Solution]

Problem 69 (In-Place Permutation). As a follow-up to the previous problem, we now develop a simple $O(n \log n)$ in-place algorithm for performing an arbitrary n-element permutation, as long as we also know the inverse of the permutation (this is true for the perfect shuffle and matrix reblocking and transposition permutations). Consider permuting an array $A[1 \ldots n]$ according to some permutation $\pi(1) \ldots \pi(n)$, where $\pi(i)$ specifies the new index into which the element $A[i]$ should be moved. We are also given the inverse of our permutation $\pi^{-1}(1) \ldots \pi^{-1}(n)$, so $\pi^{-1}(\pi(i)) = i$ and $\pi(\pi^{-1}(i)) = i$. We apply our permutation by shifting elements around its "cycles". For example, we move $A[1]$ to index $\pi(1)$, which displaces element $A[\pi(1)]$ which we then move to index $\pi(\pi(1))$, and so on, until eventually, we close a cycle and place an element back at index 1. We can trace out such a cycle starting from any element in our array, and if that element happens to have the smallest index among all elements in the cycle, we call that element the *leader* of the cycle. Our algorithm scans through all array indices $i = 1 \ldots n$ and rotates the cycle containing element i only if i is the cycle leader, thereby ensuring each cycle is rotated only once[16]. The only remaining problem is to determine if index i is the leader of its cycle. To do this, we could scan forward to $\pi(i)$, $\pi(\pi(i))$, and so on, until we either encounter an index smaller than i (in which case i was not the leader) or we return to i (in which case i was the leader). This may take too long if our permutation has long cycles, so instead we walk outward simultaneously in *both* directions, visiting $\pi(i)$ and $\pi^{-1}(i)$, then $\pi(\pi(i))$ and $\pi^{-1}(\pi^{-1}(i))$, and so on, until we again reach an index less than i or return to i. Try to prove that regardless of our permutation, this bidirectional scan results in an $O(n \log n)$ total worst-case running time for our permutation algorithm[17]. [Solution]

[16] Normally, we would ensure each cycle is rotated only once by simply marking its elements when they are rotated. However, that wouldn't give an in-place algorithm, so we use the leader trick instead.

[17] You may want to also consider how this analysis leads to an $O(n \log n)$ time solution to problem 131 as well.

